Learning Dynamics in Feedforward Neural Networks

by

Jagesh V. Shah

B. A. Sc., University of Waterloo (1992)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

Author .....
Department of Electrical Engineering and Computer Science
January 29, 1995

Certified by .....
Chi-Sang Poon
Principal Research Scientist
Thesis Supervisor

Accepted by .....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Learning Dynamics in Feedforward Neural Networks

by

Jagesh V. Shah

Submitted to the Department of Electrical Engineering and Computer Science
on January 29, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

This investigation studies the learning dynamics of artificial feedforward neural networks for the purpose of designing new algorithms and architectures with increased learning ability and computational efficiency. The concept of a system matrix for the learning algorithm is developed and the condition number of this matrix is shown to be a good indicator of convergence in backpropagation. The concept of a linearly independent internal representation is developed and used to derived the minimal number of hidden units required for exact learning. The minimization of the sum of squared error is shown to enhance the linear independence of the internal representation, thereby increasing the likelihood of exact learning. A family of architectures, the Tree-Like Architectures, are studied for potential efficacy in learning and computational efficiency. The Tree-Like Perceptron network, a member of the Tree-Like Architecture family, is useful in the context of resource limited training when compared to the Multilayer Perceptron. In addition, preliminary results show that the Tree-Like Perceptron network may enhance convergence in large-scale applications. The Tree-Like Shared network, another member of the Tree-Like Architecture family, combines the Tree-Like Perceptron with the concept of a linearly independent internal representation. The Tree-Like Shared network allows communication between decomposed subnetworks to increase convergence at the expense of parallelism. A novel two hidden layer network with a reduced parameter representation is proposed and analyzed by numerical simulation. Preliminary results indicate a dependence on the available dynamic range and precision of the computing platform.

Thesis Supervisor: Chi-Sang Poon
Title: Principal Research Scientist

Acknowledgments

I would like to thank my advisor for the time and effort he has spent contributing to my training as a researcher.

Thanks to the Canadian Natural Science and Engineering Research Council and HST for their financial support.

This research was greatly aided by the use of the CNAPS Neurocomputing platform, loaned to us by Adaptive Solutions Incorporated. Thanks to the SCOUT Consortium for the computation time on the Connection Machine CM-5 and the MIT Supercomputing Facility staff for their guidance in using the CRAY X-MP. Thank you to the consultants at the Dec/SUN Consulting Group, without whom, life would have been much more frustrating.

I would like to thank my parents and my little sister, Nisha, for their support over my many years of school and many more to come.

Many thanks to Sangeeta for her unwavering confidence in my ability.

To my family and friends back in Toronto and family and friends in the Boston area and ports all over the world, I thank you all for your words of encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Background

Artificial neural networks (ANNs) have enjoyed a great deal of success in recent years [29, 28]. They have been useful in a wide variety of tasks that require pattern recognition or feature extraction capabilities. Examples include optical character recognition, speech recognition and adaptive control, to name a few. In particular, artificial feedforward neural networks (AFNNs) have enjoyed special success because of their universal approximation capability [33] and simple and effective training algorithms [62].

Early AFNNs took the form of the single layer perceptron, originally conceived by Rosenblatt [32] in 1962. The single layer perceptron has a simple and efficient learning algorithm and a parallel computational structure (Figure 1-1), making it an excellent substrate for biological models of learning and as a tool in pattern recognition tasks.

The single layer perceptron, however, is limited in its applicability. Work by Minsky and Papert [47] and Nilsson [52] describe the requirement of linear separable input-output patterns for the single layer perceptron to successfully learn a set of patterns. This limitation could be circumvented by the addition of intermediate elements which could transform the data into a linearly separable form. It was conjectured at the time (Minsky and Papert [47], 1969) that such a multilayer perceptron, a perceptron with intermediate units, could, in principle, learn any set of input-output patterns providing the appropriate intermediate coding existed. However, the lack of an appropriate learning algorithm caused a marked decrease in neural network

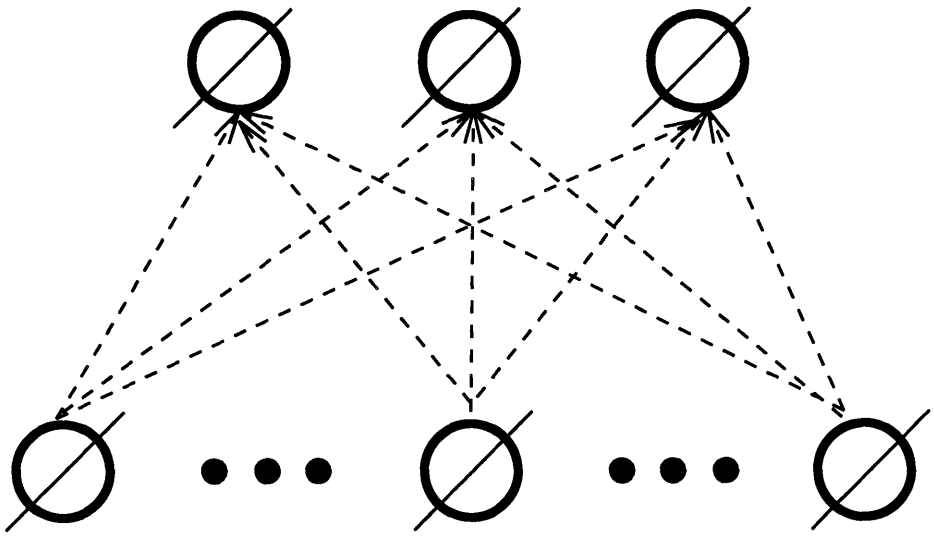


Figure 1-1: Single Layer Perceptron

research for almost 20 years [32].

In 1986, Rumelhart, Hinton and Williams [62] discovered the backpropagation of errors algorithm. This algorithm has proved to be effective in training Multilayer Perceptrons (Figure 1-2) for a variety of tasks. Unfortunately, the backpropagation algorithm does not have the guaranteed convergence results that could be shown for the single layer perceptron. Thus, training via the backpropagation algorithm can result in suboptimal solutions [11, 24, 12]. In addition, the problems of initial condition sensitivity [43, 38], long training time [72] and computational complexity [42, 71] were identified. To overcome some of these problems, such as long training time, many modifications to the backpropagation algorithm have been proposed [6, 63, 73, 17, 59, 69, 58, 10, 67].

Further work in specialized hardware [27, 35, 50, 44, 37, 46] and parallel algorithms [78, 45, 40, 21, 57], designed especially for the implementation of backpropagation, have seemingly eliminated many of the problems outlined. By having fast VLSI or parallel computer resources, the computations are faster and can be run multiple times from a variety of initial conditions to overcome the problem of suboptimal solutions and initial condition sensitivity.

Unfortunately, as the size of applications increases the computational advantages

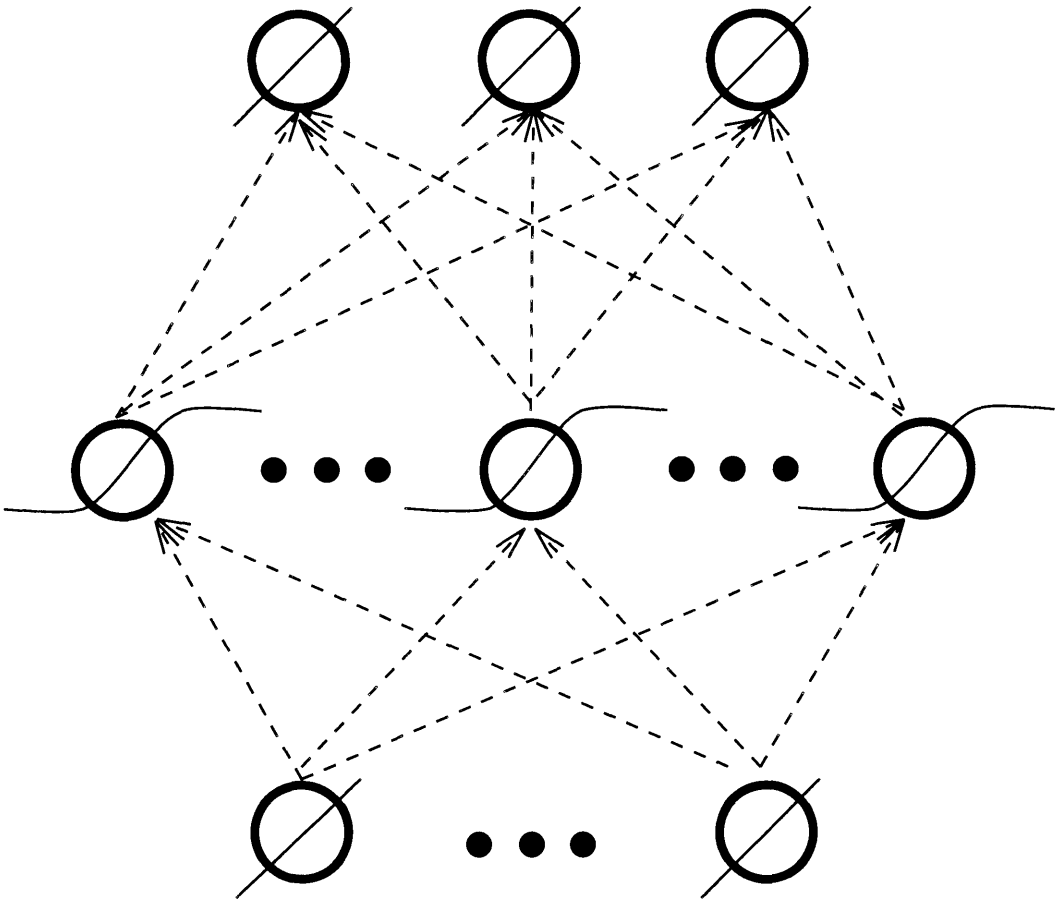


Figure 1-2: Multilayer Perceptron with one layer of hidden units with sigmoidal activation functions.

of specialized hardware and parallel algorithms are likely to become smaller because of the problems fundamental to the backpropagation algorithm and the MLP. The development of new training strategies and architectures will be required to solve these large problems within the available computational resources.

The present investigation deals with identifying important aspects of MLP learning dynamics. Using this analysis new algorithms and architectures can, in principle, be developed with faster training, smaller likelihood of suboptimal solutions and smaller computational requirements, making the available computational resources more effective at solving large AFNN tasks.

Chapter 2

Condition Number as a Convergence Indicator in Backpropagation

2.1 Introduction

Backpropagation of errors [76, 62] is the most popular training algorithm for feedforward artificial neural networks [29]. Its popularity is a result of its simplicity and easy implementation on digital computers. A major disadvantage of backpropagation is its slow convergence [42]. Presently, there exists a myriad of algorithms which increase the convergence rate of backpropagation [67, 6, 63, 73, 17]. However, many are based on ad hoc modifications which perform well under simulation of specific examples, but offer little in the way of analysis. To design algorithms which have better convergence rates it is important to understand the factors underlying learning dynamics.

This investigation reformulates the backpropagation training algorithm into a linear algebraic framework. As a result, analysis of training dynamics is simpler and linear techniques can be used to predict algorithm performance.

In section 2.2 the *system matrix*, $\mathbf{A}(\mathbf{W})$, is defined. It represents the core of this analysis. In section 2.3 the use of the condition number of a matrix as a related notion

of linear independence is discussed. The condition number of the system matrix is used to quantify the training process. Simulation results in section 2.4 illustrate the use of condition number as a convergence indicator.

In section 2.5, the matrix formulation is used to predict the performance of various algorithms. These include the use of derivative noise [17] and the use of the cross entropy cost function [67, 73]. The addition of hidden units is also analyzed for convergence performance.

2.2 Construction of the System Matrix, A(W)

This section outlines the construction of the system matrix from the basic backpropagation of errors [62] weight update formulae.

Consider a multilayer perceptron with N_I inputs, N_H hidden units and one output unit. The training set consists of N_P patterns and the training cost function is chosen to be the sum of squared error, i.e. $E(\mathbf{W}) = \frac{1}{2}\sum_{l=1}^{N_P}(t^l - z^l)^2$. where t^l is the target output for the input pattern $\mathbf{v}^l$, and z^l the corresponding network output. The superscripts denote the pattern number. The vector $\mathbf{w}$ represents the weight vector of the network. It is comprised of $\mathbf{W}_\Omega$, the weights from the hidden units to the output unit and $\mathbf{W}_\Lambda$, the weights from input to hidden nodes. Individual weights in $\mathbf{W}_\Omega$ are denoted $W_{\Omega_i}, i = 1, \ldots, N_H + 1$. Weights in $\mathbf{W}_\Lambda$ are further divided into $\mathbf{W}_{\Lambda_i}, i = 1, \ldots, N_H$, the weight vectors from the input nodes to the i^{th} hidden unit. A particular weight that connects hidden node i and input j is denoted $W_{\lambda_{i,j}}$. The $W_{\Omega_{N_H+1}}$ and $W_{\lambda_{i,N_I+1}}$ weights in $\mathbf{W}$ represent the bias weights which have an activation of unity.

As prescribed by the backpropagation algorithm the weight vector $\mathbf{w}$ is updated according to the gradient descent rule i.e. $\Delta\mathbf{W} = -\eta\nabla_{\mathbf{W}}E(\mathbf{W})$. where η represents the step size or learning rate.

The weight update rules can be written explicitly as

$$\Delta W_{\Omega_i} = \eta\sum_{l=1}^{P} f'(\overline{y_o^l})y_i^l(t^l - z^l) \tag{2.1}$$

$$\Delta W_{\lambda_{i,j}} = \eta \sum_{l=1}^{P} f'(\overline{v_i^l}) v_j^l \cdot f'(\overline{y_o^l}) W_{\Omega_i}(t^l - z^l) \tag{2.2}$$

where $\overline{v_i^l}$ is the activation for hidden node i and $\overline{y_o^l}$ is the activation for the output node. y_i^l is the output of hidden node i and v_j^l is the j^{th} component of the input pattern. The terms $y_{N_H+1}^l$ and $v_{N_I+1}^l$ are unity, for the update of the bias weights. In addition, $f(\cdot)$ is the nodal activation function and $f'(\cdot)$ its first derivative. Throughout this discussion it is assumed that $f(\cdot)$ is at least once continuously differentiable, bounded and monotonically increasing. The sigmoid function $f(x) = 1/(1+e^{-x})$ is an example of such a function. This function is used in all simulations presented.

The update equations can be interpreted as a set of linear equations in the term $(t^l - z^l)$. Let $e^l(\mathbf{W}) = t^l - z^l$ and $\mathbf{e}(\mathbf{W}) = [e^1(\mathbf{W}) \ldots e^P(\mathbf{W})]^T$. $\mathbf{e}(\mathbf{W})$ is the error vector. The weight update equations can now be rewritten as

$$\Delta \mathbf{W}_{\mathbf{\Omega}} = \eta \mathbf{A}_{\mathbf{\Omega}}(\mathbf{W})\mathbf{e}(\mathbf{W}) \tag{2.3}$$

$$\text{where } [\mathbf{A}_{\mathbf{\Omega}}(\mathbf{W})]_{qr} = [f'(\overline{y_o^r}) y_q^r]_{qr} \tag{2.4}$$

$$\Delta \mathbf{W}_{\mathbf{\Lambda}_i} = \eta \mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W})\mathbf{e}(\mathbf{W}) \tag{2.5}$$

$$\text{where } [\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W})]_{qr} = [f'(\overline{v_i^r}) v_q^r f'(\overline{y_o^r}) W_{\Omega_i}]_{qr} \tag{2.6}$$

$[\mathbf{A}]_{qr} = [a_{qr}]_{qr}$ is used to denote a matrix $\mathbf{A}$, with element a_{qr} at row q and column r. These equations can be further accumulated into one matrix equation for the simultaneous update of all the weights in $\mathbf{W}$. When equations 2.4 and 2.6 are combined, the result is

$$\Delta \mathbf{W} = \eta \mathbf{A}(\mathbf{W})\mathbf{e}(\mathbf{W}), \text{ where } \mathbf{A}(\mathbf{W}) = \left[\begin{array}{c} \mathbf{A}_{\mathbf{\Omega}}(\mathbf{W}) \\ \hline \mathbf{A}_{\mathbf{\Lambda}_1}(\mathbf{W}) \\ \vdots \\ \mathbf{A}_{\mathbf{\Lambda}_{N_H}}(\mathbf{W}) \end{array}\right] = \left[\begin{array}{c} \mathbf{A}_{\mathbf{\Omega}}(\mathbf{W}) \\ \hline \mathbf{A}_{\mathbf{\Lambda}}(\mathbf{W}) \end{array}\right] \tag{2.7}$$

$\mathbf{A}(\mathbf{W})$ is the system matrix.

We assume that $N_P \leq (N_H + 1) + N_H(N_I + 1)$, that is, the number of training

patterns is less than the number of weights and biases in the network. This assumption holds for many applications in which artificial neural networks have been used.

2.3 Condition Number of A(W) and Network Convergence

This section discusses the use of the condition number as a relative measure of rank in a matrix. The result is an interpretation of the system matrix which is illustrated through simulation.

The condition number of a matrix is the ratio of the largest and smallest singular values of a matrix [39]. A large condition number corresponds to a matrix which has columns which are *nearly* linearly dependent. This implies that rank of the matrix is *nearly* deficient. The condition number provides *relative* information about the linear dependence of the columns of a matrix rather than the *absolute* information given by the rank (i.e. either dependent or independent).

Using this linear dependence approach equation 2.7 can be rewritten as

$$\Delta \mathbf{W} = \sum_{l=1}^{P} \mathbf{a}^l(\mathbf{W}) e^l(\mathbf{W}) \tag{2.8}$$

where $\mathbf{a}^l(\mathbf{W})$ is the l^{th} column of $\mathbf{A}(\mathbf{W})$ and $e^l(\mathbf{W})$ is the scalar error for the l^{th} pattern. The term $\mathbf{a}^l(\mathbf{W})e^l(\mathbf{W})$ represents the gradient of the cost function for the l^{th} pattern. Thus equation 2.8 is the vector sum of N_P gradient vectors.

The linear dependence of the columns of $\mathbf{A}(\mathbf{W})$ (i.e. the rank) determines the number of directions that $\Delta \mathbf{W}$ can take. If the rank of $\mathbf{A}(\mathbf{W})$ is deficient, then the directions of $\Delta \mathbf{W}$ are restricted, in fact there exists a subspace of error vectors, $\mathbf{e}(\mathbf{W}) \neq \mathbf{0}$, which will result in $\Delta \mathbf{W} = \mathbf{0}$. The algorithm has become stuck in a local minimum. If, however, the rank of $\mathbf{A}(\mathbf{W})$ is full, then $\Delta \mathbf{W} \neq \mathbf{0}$ is guaranteed for $\mathbf{e}(\mathbf{W}) \neq \mathbf{0}$ and the training error continues to decrease. Note that if $\mathbf{e}(\mathbf{W}) = \mathbf{0}$ the global solution has been reached. For this analysis strict gradient descent is always guaranteed, thus there is no possibility of limit cycle behaviour.

Because of the continuous nature of the activation functions, it is unlikely that the system matrix will lose rank. However in the neighbourhood of a local minima, the columns of $\mathbf{A}(\mathbf{W})$ will become *nearly* linearly dependent. Thus it is appropriate to use the condition number as a measure of the linear dependence of the columns of $\mathbf{A}(\mathbf{W})$.

2.4 Simulation Examples

To study the linear dependence of the columns of $\mathbf{A}(\mathbf{W})$, the condition number of the system matrix and the training error of an Exclusive-Or (one layer of two hidden units) network were tracked (Figure 2-1). The graphs show that the condition number and error fall together. This is a result of the columns of $\mathbf{A}(\mathbf{W})$ becoming *more* linearly independent. In trials where convergence was not attained the condition numbers remained higher than those of the converged trials. Extended simulation runs revealed that high condition numbers may be associated with local minima. Note that one of the trials showed an increase in condition number with continued training and thus may be stuck in a local minimum.

Final solution optimality was also characterized using the condition number. A set of trials were run for a fixed duration. The condition number of the system matrix was then calculated for each trial. The convergence criteria was a sum of squared error less than 0.05. Five hundred Exclusive-Or networks (as above) were simulated. Figure 2-2 shows the results of the simulation. Of the 500 trials, 299 converged and 201 did not. The bars represent the mean condition number of the final system matrix. Note that for the converged trials the final condition number of the system matrix is much smaller. This indicates a strong correlation between network convergence to a globally optimal solution and a low system matrix condition number.

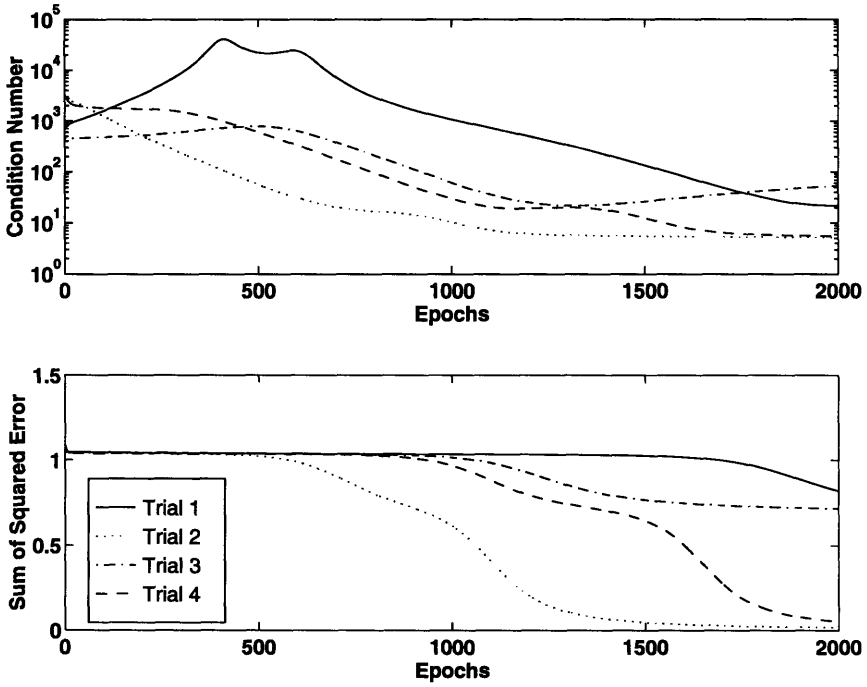


Figure 2-1: Condition Number and Network Error Trajectories

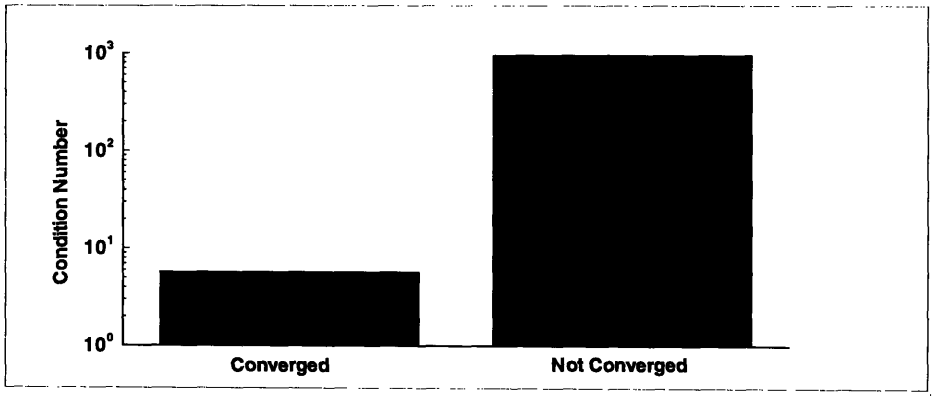


Figure 2-2: Mean condition numbers of converged and unconverged trials after 10000 epochs

2.5 Enhancing Convergence by Conditioning A(W)

Simulation results have indicated that there is a strong correlation between a low condition number and final solution optimality. In this section algorithms that exploit this relationship for better training performance are discussed.

2.5.1 Decreasing the Condition Number of A(W)

Before examining the algorithms, it is useful to outline some of the properties of the system matrix that they exploit.

In examining the rank of the system matrix only the columns were considered for linear independence. The rank calculation can also be viewed as an examination of linearly independent rows of the system matrix. In particular, many of the rows of the submatrix $\mathbf{A}_{\mathbf{\Lambda}}(\mathbf{W})$ are linearly dependent. To see this note Equation 2.6. The l^{th} column of $\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W})$ shares the term $f'(\overline{v_i^l})f'(\overline{y_o^l})W_{\Omega_i}$. Since $f'(\cdot) > 0$ for any finite argument, $f'(\overline{v_i^l})f'(\overline{y_o^l})W_{\Omega_i}$ can be eliminated from each column without changing the rank of $\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W})$. Thus $\mathcal{R}(\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W})) = \mathcal{R}([v_q^r]_{qr})$ where $\mathcal{R}(\cdot)$ is the rank operator. Note that the simplified matrix is independent of i, implying $\mathcal{R}(\mathbf{A}_{\mathbf{\Lambda}}(\mathbf{W})) = \mathcal{R}(\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W}))$. Thus, many of the rows of $\mathcal{R}(\mathbf{A}_{\mathbf{\Lambda}}(\mathbf{W}))$ do not contribute to increasing the condition number of $\mathbf{A}(\mathbf{W})$. This is an inherent redundancy in backpropagation that can be exploited to increase convergence performance.

This simplification can also be performed for $\mathbf{A}_{\mathbf{\Omega}}(\mathbf{W})$. However, the resulting rank is the same as that of $\mathbf{A}_{\mathbf{\Omega}}(\mathbf{W})$. This is because the terms of the simplified matrix are functions of the network weights which change over the training period. Thus no rows can be eliminated due to linearly dependency.

2.5.2 Derivative Noise

One ad hoc method of achieving the modulation of the elements of $\mathbf{A}_{\mathbf{\Lambda}}(\mathbf{W})$ is to randomize them by adding noise. If the terms in the matrix are disturbed using noise more rows can potentially contribute to the condition number. At the same time, however, the guarantee of strict gradient descent is lost. As a result this method can

be very sensitive to the magnitude of noise used. In spite of this, the noise method has been shown to prove effective by Fahlman [17] who used the noise to prevent saturation of the derivative of the activation function. When a small amount of noise was added to the derivatives, the networks converged faster and were less likely to get stuck in local minima. In other words, by keeping more rows linearly independent, the algorithm could proceed without being impeded by local minima.

2.5.3 Cross Entropy versus Sum of Squared Error

Another method of increasing convergence performance is to increase the magnitude of the weight update vector. Thus a larger step is taken down the error surface during each weight update, resulting in faster convergence.

The system matrix, $\mathbf{A}(\mathbf{W})$, can be simplified by eliminating the term $f'(\overline{y_o^l})$ from each column. This is in direct analogy with the simplification of $\mathcal{R}(\mathbf{A}_{\mathbf{\Lambda}_i}(\mathbf{W}))$. The sigmoidal activation function, $f(x) = 1/(1+e^{-x})$, has the property that $0 < f'(x) < 1$. That is $f'(\overline{y_o^l})$ has the effect of attenuating the matrix elements and thus the magnitude of the weight update vector. The removal of this term should accelerate convergence.

This approach has been used in training algorithms that have a cross entropy [67] type cost function. The cross entropy cost function has the form

$$E(\mathbf{w}) = \sum_{l=1}^{P} \left\{ t^l \ln \frac{t^l}{o^l} + (1-t^l) \ln \frac{(1-t^l)}{(1-o^l)} \right\} \tag{2.9}$$

for a single output network.

When weight update equations are calculated for this cost function, the result is a system matrix without the $f'(\overline{y_o^l})$ term. These algorithms yield faster convergence when compared with the sum of squared error cost function as shown in [67] and [73].

2.5.4 Increasing the Number of Hidden Units

The condition number of $\mathbf{A}(\mathbf{W})$ can also be increased by increasing the number of rows. In particular, adding rows to $\mathbf{A}_{\mathbf{\Omega}}(\mathbf{W})$ is equivalent to adding more hidden units

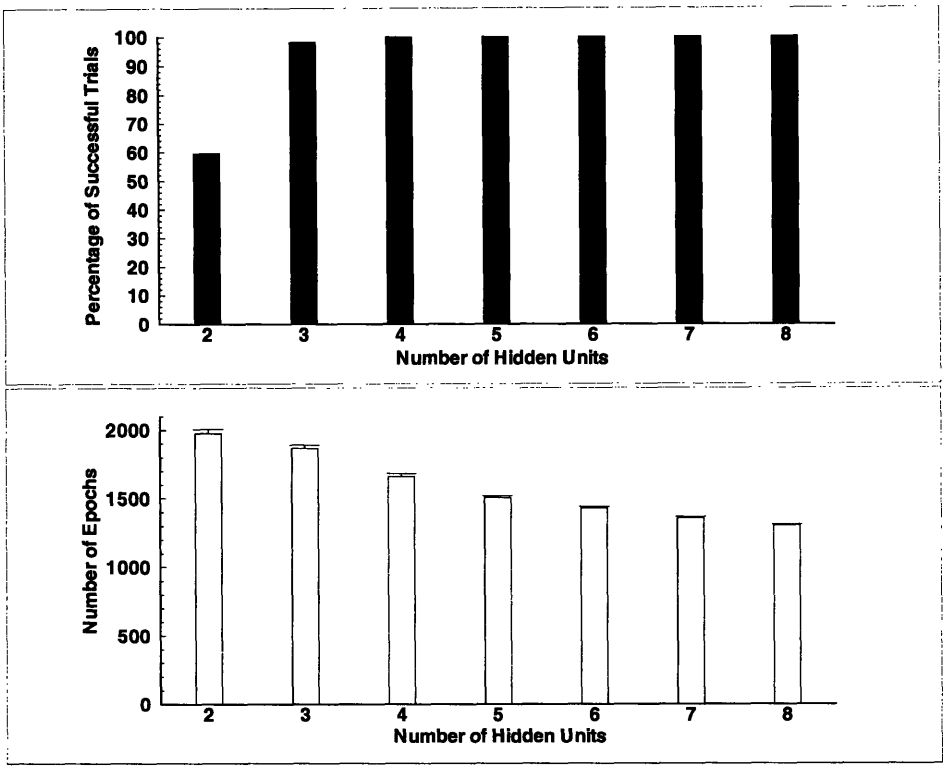


Figure 2-3: Effect of Number of Hidden Units on Network Convergence

to the network. It follows that more hidden nodes should increase the possibility of obtaining a better conditioned system matrix. A similar redundancy approach is taken by Izui and Pentland [36]. They show that when redundant nodes (input and hidden) are added to the network they speed up convergence. In figure 2-3 the number of converged networks is plotted for the Exclusive-Or example with varying hidden layer sizes. Note the increase in the percentage of converged networks as the number of hidden units is increased. In fact, with four or more hidden units all the trials converged. Figure 2-3 also shows the number of epochs required to achieve the convergence criterion. As the number of hidden units increases the number of epochs needed to achieve convergence decreases.

2.6 Discussion

In this investigation, a linear algebraic formulation has been described that is useful in predicting the convergence performance of algorithms that are based on backprop-

agation of errors.

Simulations showed the correlation between the training error and system matrix condition number trajectories.

The enhanced performance of algorithms with derivative noise [17] and cross entropy cost functions [67, 73] were predicted using the matrix formulation. In addition, improved performance due to additional hidden units was predicted by the formulation and illustrated through simulation. The approach is similar to that found in [36].

Chapter 3

Linear Independence of Internal Representations in Multilayer Perceptrons

3.1 Introduction

The shortcomings of the single layer perceptron were well described by Minsky and Papert [47] and Nilsson [52]. But these investigators were also aware that a recoding of the inputs could result in a set of patterns that was learnable. That is, by transforming the input patterns into some intermediate values, the input patterns could be augmented to give a new set of patterns which the single layer perceptron could learn. Hence, the notion of an internal representation was established.

In current terminology the internal representation refers specifically to the outputs of the hidden units of a multilayer network. In analogy with Minsky and Papert [47] and Nilsson [52], the internal representation is considered a coding of the input patterns into a form which is easily *learnable* [62].

The role of the internal representation is seemingly to increase the *learnability* of the training patterns. Unfortunately, although the notion of internal representation is widely recognized, the underlying mechanism of the increased *learnability* is not

completely understood.

Investigations into threshold MLPs (i.e. having threshold hidden unit activation functions) have identified important features of the internal representation, such as linear separability, which are necessary for successful training (i.e. exact learning) of a set of input-output patterns [52, 47, 48, 34, 70]. Followup work has resulted in algorithms that train threshold MLPs by modifying the internal representation of a threshold MLP directly to achieve exact learning [70, 26].

Unfortunately, threshold MLPs are not widely used in practice. This is because threshold activation functions are not continuously differentiable – a condition that is necessary in many training algorithms (e.g. backpropagation). The sigmoid function, or soft threshold, is a differentiable approximation to the threshold activation. Results due to Sontag [68] show that MLPs with sigmoidal activation functions have increased capacity over threshold MLPs. As a result, sigmoidal activation functions are used widely in MLP applications.

The focus of this investigation is to identify important features of internal representations of sigmoidal MLPs (i.e. having sigmoidal hidden unit activation functions) necessary for successful training. In particular, the linear independence of the internal representation is studied. Work by Webb and Lowe [75] and Fujita [22, 23] have noted that linear independence is important in the internal representation of successfully trained MLPs.

In addition, the linear independence of the internal representation decreases the condition number of the system matrix, $\mathbf{A}(\mathbf{W})$, introduced in Chapter 2. The decrease in the condition number of the system matrix is correlated with convergence to zero error.

This investigation describes rigorously the ability of sigmoidal activation functions to produce a linearly independent internal representation. This result is used to derive the minimum number of hidden units required for successful training of an arbitrary set of patterns. Furthermore, a relationship between linear independence and current strategies for MLP training is discovered. The relationship indicates that current training strategies implicitly modify the linear independence of the internal

representation to increase the likelihood of achieving exact training.

3.2 Internal Representation in Sigmoidal MLPs

This section outlines a structural definition of the internal representation that will be used in this investigation. The formulation of the MLP and its internal representation in vector form is important in understanding the necessity of linear independence for successful training.

Consider an MLP network having N_I input units, one layer of N_H hidden units and one output unit. Let the hidden unit activation functions be hyperbolic tangent (tanh) sigmoidal nonlinearities and the output units be linear. Work by Hornik [33] shows that one layer of sigmoidal hidden units is sufficient to approximate any continuous function. The hyperbolic tangent activation function is used throughout this investigation because of its symmetric properties which makes the results less tedious. All results, however, can be extended to any sigmoidal function. Linear output units are used in this investigation to make explicit use of the *linear* combination of the vectors of the internal representation. However, the results can be extended to any bijective (i.e. 1:1) output activation function. The use of sigmoidal output units in practice may be due to the use of binary target patterns and the limited dynamic range of many computer platforms. Note that the use of a linear output node does not bound the output of the network, thereby making the output difficult to represent on machines with a finite dynamic range. Note that networks with a single output are considered for the bulk of the investigation for simplicity of exposition. Important results are extended to multi-output networks.

3.2.1 Computation of Internal Representation from Input

Let the training data for the MLP be a set of N_P input-output patterns. Assume that the training set has *distinct* input patterns. That is, for each input pattern there is only one corresponding output pattern. This condition prevents the possibility of conflicting input-output patterns, which makes exact learning impossible.

The internal representation is the output of the hidden units over the presentation of the training input patterns. The transformation of the inputs to the associated hidden outputs over the entire set of training patterns can be represented as :

$$\begin{bmatrix} [\mathbf{input}_1]_1 & \dots & [\mathbf{input}_1]_{N_I} \\ \vdots & & \vdots \\ \left[\mathbf{input}_{N_P}\right]_1 & \dots & \left[\mathbf{input}_{N_P}\right]_{N_I} \end{bmatrix} \\ \downarrow \\ \begin{bmatrix} [\mathbf{hidden}_1]_1 & \dots & [\mathbf{hidden}_1]_{N_H} \\ \vdots & & \vdots \\ [\mathbf{hidden}_{N_P}]_1 & \dots & [\mathbf{hidden}_{N_P}]_{N_H} \end{bmatrix} \tag{3.1}$$

where $\left[\mathbf{input}_p\right]_i$ is the i^{th} element of the p^{th} input pattern and $[\mathbf{hidden}_p]_j$, the j^{th} element of the transformed input pattern.

Equation 3.1 shows the transformation of every element of the input data into the internal representation. This matrix is similar to the constructions used in Fujita [22] and Webb and Lowe [75].

The matrix of hidden units outputs in Equation 3.1 can be interpreted in two directions, pattern-wise (i.e row-by-row) or node-wise (i.e. column-by-column).

The *pattern-wise* analysis isolates one input pattern and the corresponding hidden unit outputs. For example, the p^{th} input pattern is transformed as :

$$\begin{bmatrix} \left[\mathbf{input}_p\right]_1 & \dots & \left[\mathbf{input}_p\right]_{N_I} \end{bmatrix} \\ \downarrow \\ \begin{bmatrix} [\mathbf{hidden}_p]_1 & \dots & [\mathbf{hidden}_p]_{N_H} \end{bmatrix} . \tag{3.2}$$

Thus, the transformation on each input pattern can be studied in detail. This method of analysis is used frequently to study what features a network has extracted from an input data set [64, 25]. That is, by studying the transformation due to a previously trained network on each input pattern, the elements of the input which are important in producing the target can be isolated. This is essentially the notion of an internal

representation described by Minsky and Papert [47] – a recoding of the input patterns for easier *learnability.*

The *node-wise* analysis differs from the pattern-wise analysis in that it uses all the input patterns and isolates each hidden node output. That is,

$$\begin{bmatrix} [\mathbf{input}_1]_1 & \dots & [\mathbf{input}_1]_{N_I} \\ \vdots & & \vdots \\ [\mathbf{input}_{N_P}]_1 & \dots & [\mathbf{input}_{N_P}]_{N_I} \end{bmatrix} \longrightarrow \begin{bmatrix} [\mathbf{hidden}_1]_j \\ \vdots \\ [\mathbf{hidden}_{N_P}]_j \end{bmatrix} \tag{3.3}$$

is the transformation of the input patterns to the output of the j^{th} hidden unit. By studying all the patterns at once, all the output patterns can be generated simultaneously. Moreover, all the output patterns can be checked against all the target patterns simultaneously to see if the network has learned the patterns exactly. The node-wise analysis simplifies the study of exact learning.

Note that the vector of the hidden unit outputs, i.e. the internal representation, combine linearly to form the output patterns. Since the hidden unit outputs (or vectors) combine linearly, the study of their linear independence is important. This node-wise method is similar to analyses found in many other investigations [75, 22, 24, 11, 55, 54, 5]. These investigations do not directly address the production of linear independence in the node-wise internal representation. Although, Webb and Lowe [75], Fujita [22] and Blum [11] discuss the presence of linearly independent node-wise vectors in networks achieving exact learning.

Define the vector,

$$\mathbf{y}_j = \begin{bmatrix} [\mathbf{hidden}_1]_j \\ \vdots \\ [\mathbf{hidden}_{N_P}]_j \end{bmatrix}$$

to be the j^{th} node-wise vector of the internal representation. As will be shown, the vectors of the internal representation determine the space of possible outputs of the MLP, and the larger the space, the greater the likelihood of having achieved exact learning.

To consider the computation of the MLP using the node-wise internal representation, the input and output patterns must also be changed into a node-wise orientation. Define the i^{th} node-wise *input vector* to be:

$$\mathbf{v}_i = \begin{bmatrix} [\mathbf{input}_1]_i \\ \vdots \\ \left[\mathbf{input}_{N_P}\right]_i \end{bmatrix}.$$

That is, the i^{th} node-wise input vector is the input to the i^{th} input node over the training set. Similarly, define the j^{th} node-wise *hidden vector*, denoted $\mathbf{x}_j$, to be the input to the j^{th} hidden unit over the training set. The node-wise hidden vectors are linear combinations of the node-wise input vectors. That is,

$$\mathbf{x}_j^T = \left[\mathbf{v}_1 \,\middle|\, \ldots \,\middle|\, \mathbf{v}_{N_I} \,\middle|\, \mathbf{1} \right] \cdot \mathbf{W}_{\Lambda_j} \tag{3.4}$$

where $\mathbf{W}_{\Lambda_j}$ is the column vector of weights associated with the j^{th} hidden unit and $\mathbf{1}$ is the N_P-dimensional vector of ones representing the thresholds. $\mathbf{A}^T$ is the transpose of the matrix $\mathbf{A}$. Equation 3.4 represents the first layer of computation in the MLP.

The application of the sigmoidal activation functions to the node-wise hidden vectors results in the vectors that form the node-wise internal representation. That is,

$$\mathbf{y}_j = \mathbf{tanh}(\mathbf{x}_j) \tag{3.5}$$

where the vector function **tanh** is the element-by-element hyperbolic tangent. The set of vectors $\{\mathbf{y}_1, \ldots, \mathbf{y}_{N_H}\}$ form the node-wise internal representation.

3.2.2 Computation of Output from Internal Representation

The final layer of computation in the MLP is the production of the output. The node-wise *output vector*, denoted $\mathbf{z}$, is the output of the single output unit of the MLP over the training set. Define $\mathbf{T}$ to be the column vector of target outputs from the training set, the node-wise *target vector*. Exact learning is achieved when $\mathbf{z} \equiv \mathbf{T}$.

Let the *activation vectors* be those vectors which contribute to the output vector, $\mathbf{z}$. As noted in the three cases below, the activation vectors can consist of the input vectors, vectors of the internal representation or both. For exact learning to be achieved, the activation vectors must span a subspace which includes the target vector, $\mathbf{T}$. Each of the three cases shown below describe the output computation for a specified architecture. Each architecture contributes different vectors to the computation of the output and therefore will have different spaces spanned at the output. As a result, conditions on exact learning can be described for each architecture.

Case I : No Hidden Units (i.e. $N_H = 0$)

Consider the case when $N_H = 0$, that is, the single layer perceptron (Figure 3-1). The computation for the output is :

$$\mathbf{z}^T = \left[\mathbf{v}_1 \,\middle|\, \ldots \,\middle|\, \mathbf{v}_{N_I} \,\middle|\, \mathbf{1} \right] \cdot \mathbf{W}_\Omega \tag{3.6}$$

where $\mathbf{W}_\Omega$ is the column vector of hidden to output weights. If the training patterns are to be learned exactly then,

$$\mathbf{T} \in \mathcal{RA}\left(\left[\mathbf{v}_1 \,\middle|\, \ldots \,\middle|\, \mathbf{v}_{N_I} \,\middle|\, \mathbf{1} \right] \right) \tag{3.7}$$

must hold, where $\mathcal{RA}(\mathbf{A})$ is the range space of the matrix $\mathbf{A}$. If there are nonlinear relationships in the input-output data, however, Equation 3.7 will not hold and the single layer perceptron cannot learn the patterns exactly [47, 52]. Thus, the architecture is restricted to learning linearly separable patterns [47, 52]. The next case shows how the addition of hidden units can allow the network to learn even nonlinear relationships.

Case II : N_H Hidden Units

By adding hidden units between the input nodes and output unit, Equation 3.6 becomes :

$$\mathbf{z}^T = \left[\mathbf{y}_1 \,\middle|\, \ldots \,\middle|\, \mathbf{y}_{N_H} \,\middle|\, \mathbf{1} \right] \cdot \mathbf{W}_\Omega. \tag{3.8}$$

This computation corresponds to the conventional MLP (Figure 3-1). The condition

of exact learning now becomes:

$$\mathbf{T} \in \mathcal{RA}\left(\left[\ \mathbf{y}_1 \ \middle|\ \ldots \ \middle|\ \mathbf{y}_{N_H} \ \middle|\ \mathbf{1}\ \right]\right). \tag{3.9}$$

Under the condition given in Equation 3.9 the internal representation must produce the required vectors to span a subspace that includes the target vector, $\mathbf{T}$, as a subset. If Equation 3.7 does not hold, i.e. there are nonlinear relationships in the data, then some of the $\mathbf{y}_i$'s must be linearly independent of the input vectors. In fact, even if the relationship is linear, i.e. Equation 3.7 holds, the $\mathbf{y}_i$'s may be linearly independent since the $\mathbf{v}_i$'s do not contribute to the output. Section 3.3 shows that the sigmoidal hidden unit can produce vectors that are linearly independent of the input vectors. In principle, this architecture can learn the training patterns exactly. If the required space can be constructed from linear and nonlinear relationships, this architecture has the potential disadvantage of having to reproduce linear combinations of the input vectors, redundantly. The next case discusses the situation when the direct input to output connections are added to the architecture.

Case III : N_H Hidden Units, with direct input to output connections

By adding the direct input to output connections to the architecture (Figure 3-1), the computation of the output becomes :

$$\mathbf{z}^T = \left[\ \mathbf{y}_1 \ \middle|\ \ldots \ \middle|\ \mathbf{y}_{N_H} \ \middle|\ \mathbf{v}_1 \ \middle|\ \ldots \ \middle|\ \mathbf{v}_{N_I} \ \middle|\ \mathbf{1}\ \right] \cdot \mathbf{W}_\Omega. \tag{3.10}$$

Note that now $\mathbf{W}_\Omega$ consists of both hidden to output and input to output weights. The condition for exact learning now changes to :

$$\mathbf{T} \in \mathcal{RA}\left(\left[\ \mathbf{y}_1 \ \middle|\ \ldots \ \middle|\ \mathbf{y}_{N_H} \ \middle|\ \mathbf{v}_1 \ \middle|\ \ldots \ \middle|\ \mathbf{v}_{N_I} \ \middle|\ \mathbf{1}\ \right]\right). \tag{3.11}$$

As in Case II, if there is some underlying nonlinear relationship (i.e. Equation 3.7 does not hold), some of the $\mathbf{y}_i$'s must be linearly independent of the input vectors for Equation 3.11 to hold. This architecture has an advantage over Case II, in that any linear relationship in the data can be produced via the direct connections, potentially

reducing the number of hidden units required. This is shown formally in Section 3.3 where the minimum number of hidden units required for successful training is derived.

These three cases considered above demonstrate that exact learning for nonlinear relationships is guaranteed only when:

1. Some of the vectors of the internal representation are linearly independent of the input vectors

2. The vectors of the internal representation are included in the computation of the output space.

3. The target vector, **T**, is a subset of the space spanned by the vectors of the internal representation (and the input vectors, for Case III).

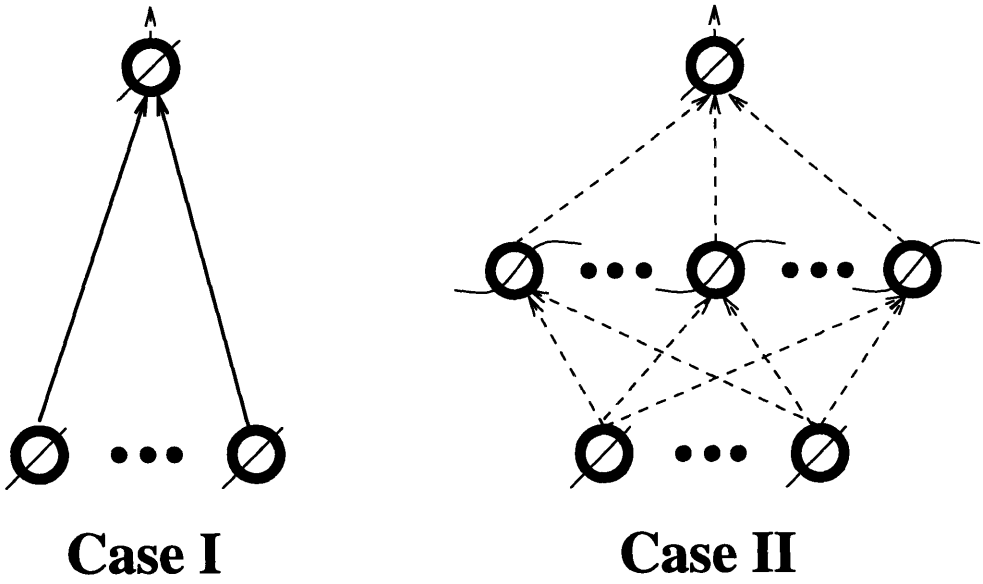
These conditions establish the necessity of a linearly independent internal representation for exact learning in an MLP. Note that these conditions also hold for multi-output systems. In the multi-output case, the target vector for each output must lie in the subspace spanned by the activation vectors.

The next section describes the ability of the MLP to produce the required linearly independent vectors of the internal representation and the number of vectors needed for exact learning.

3.3 Minimum Number of Hidden Units

In this section the ability of sigmoidal hidden units to produce linearly independent vectors is described. The mathematical foundation for such a description is developed rigorously. These results are fundamental to establishing the minimum number of hidden units required for successful training of an arbitrary set of patterns.

Sigmoidal MLPs are generally more difficult to analyze than threshold MLPs. Due to the discrete nature of threshold units, analysis is combinatorial [48] or in terms of convex polyhedra [34] created by the threshold MLP's separating surfaces. Sigmoidal units have a continuum of possible outputs between two real numbers. The separating surfaces created by the sigmoidal MLP are highly irregular and thus counting



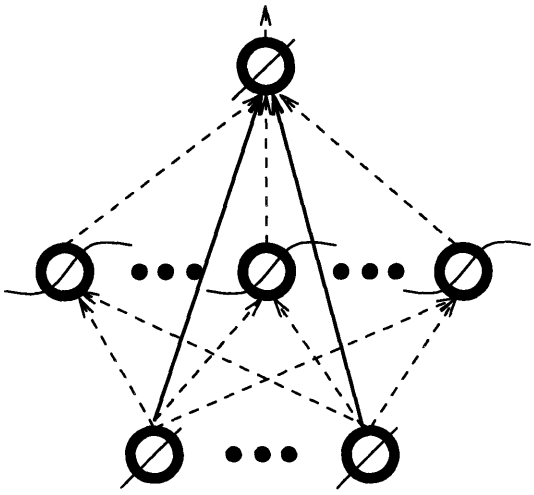


Figure 3-1: Architectures of a Multilayer Perceptron. The figure shows the three architectures corresponding to the cases described in the text. Case I is a single layer perceptron, Case II is a conventional multilayer perceptron and Case III is an MLP with direct input to output connections.

arguments fail. The result is that proofs for the continuous activation functions tend to be more subtle and tedious.

Main results appear in the text of this investigation; however, many of the proofs are placed in Appendix A.

3.3.1 Linearly Independent Vectors from a Sigmoidal Activation Function

Studying the properties of a single sigmoidal hidden unit is an important step in understanding the functional role of hidden units in MLP networks. From Equation 3.11 it is clear that in order to learn an arbitrary target vector, the hidden units *must* produce vectors which are linearly independent of the input vectors. The following technical result shows that a single tanh nonlinearity can produce the vectors required to form a complete basis from a single vector. In other words, the nonlinear transformation can span an N-dimensional *space* with a single N-dimensional *vector.*

Lemma 3.1 *Let* $\mathbf{u} \in \Re^N$ *be any non-zero vector of distinct elements, i.e.*

$$[\mathbf{u}]_i \neq [\mathbf{u}]_j \, , \; i \neq j, \; i = 1, \ldots, N, j = 1, \ldots, N,$$

Then for each nonzero vector $\mathbf{s} \in \Re^N, \exists \, a, b \in \Re$ *such that*

$$\mathbf{s}^T \mathbf{tanh}(a\mathbf{u} + b\mathbf{1}) > 0.$$

Specifically, Lemma 3.1 shows that the vectors generated by the **tanh** function as a and b vary over $\Re$ are not confined to a lower dimensional subspace of $\Re^N$. Equivalently, there is a group of N vectors from the image space of the **tanh** function that form a complete basis for $\Re^N$. This is stated more formally in the following corollary.

Corollary 3.1 *For every nonzero vector* $\mathbf{u} \in \Re^N$ *of distinct elements,* $\exists$ *ordered pairs* (a_i, b_i), $a_i \in \Re, b_i \in \Re, i = 1, \ldots, N$ *such that*

$$\{\mathbf{tanh}(a_1\mathbf{u} + b_1\mathbf{1}), \ldots, \mathbf{tanh}(a_N\mathbf{u} + b_N\mathbf{1})\}$$

forms a complete basis of $\Re^N$.

Lemma 3.1 is analogous to results due to Oh and Lee [53]. Oh and Lee show that the nonlinear function of the sum of two random variables is less linearly correlated with respect to the original random variables. That is, the output is *more* linearly independent of the input random variables. Lemma 3.1 shows a similar result for deterministic inputs.

3.3.2 Minimum Number of Hidden Units

In this section, the ability of the sigmoidal activation function to produce linearly independent vectors is used to establish the minimum number of hidden units required for successful training of an arbitrary training set.

In the context of MLP networks, Corollary 3.1 can be used to show that a one-hidden layer MLP with $N_P - 1$ sigmoidal hidden units and *without* direct input to output connections can learn any set of N_P one-dimensional input-output patterns with distinct input patterns (discussed in Section 3.2 Case II). This is stated formally in Theorem 3.1, which corresponds the worst case in hidden vector diversity.

Theorem 3.1 *Given a set of* N_P **single input, single output** *patterns with distinct input patterns there exists a network with* $N_P - 1$ *hidden units* without *direct input to output connections which can learn the patterns exactly.*

Proof:

> The input patterns form a vector of distinct elements, $\mathbf{u}$, and the bias inputs to each hidden unit form a vector of unit elements, $\mathbf{1}$. From Corollary 3.1 a complete basis of vectors can be generated. One of these vectors

can be replaced with the threshold vector of the output unit preserving the complete basis. Thus, there are $N_P - 1$ vectors generated by hidden units and one supplied by the threshold of the output unit that form a complete basis. As a result, there exist some linear combination of these basis vectors which will yield the target.

□

The one input case represents the worst case for linear independence generation. That is, with multiple input vectors the input space from which to extract the complete basis is larger. Lemma 3.2 guarantees that by using multiple input vectors, a vector of distinct elements can be generated.

Lemma 3.2 *Given a set of input vectors,* $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$*, from distinct input patterns,* $\exists\ \{a_1, \ldots, a_k\}$ *such that*

$$\mathbf{u} = \sum_{i=1}^{k} a_i \mathbf{s}_i$$

has distinct elements.

A similar result is due to Sontag [68] in which the equivalence of single input and multiple input systems is shown.

Using Lemma 3.2, Theorem 3.1 can be extended to networks with multiple inputs.

Theorem 3.2 *Given a training set of* N_P **multi-input, single output** *patterns (with distinct input patterns), a one-hidden-layer perceptron with* $N_P - 1$ tanh *hidden units and* without *direct connections from input to output can learn the patterns exactly.*

Proof:

Lemma 3.2 states that there is some linear combination of the input vectors that result in a vector of distinct elements. Since each hidden vector, $\mathbf{x}_i$, is simply a linear combination of the input vectors (Equation 3.4), each $\mathbf{x}_i$ can be trivially constructed to be the same vector of

distinct elements. As a result, the multi-input system has been reduced to a single input system and can be solved using $N_P - 1$ hidden units as per Theorem 3.1.

□

It is important to note that Theorem 3.2 applies to MLP networks *without* input to output connections. The computation of the output for this type of network is a function of the vectors of the internal representation only (Equation 3.9) as opposed to Equation 3.11 where the input vectors are also present. If direct connections between input and output are present (as described in Section 3.2.2 Case III) the number of linearly independent vectors of the internal representation required to produce a complete basis is smaller. Thus fewer hidden units will be required. In fact, the number of hidden units drops by the number of linearly independent input vectors. With this in mind the following proposition is proven.

Proposition 3.1 *To learn an arbitrary set of N_P* **multi-input single output** *patterns with distinct input patterns with an MLP network with direct input to output connections,*

$$N_P - \mathcal{R}\left(\left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_{N_I} & \mathbf{1} \end{array}\right]\right)$$

tanh *hidden units are required. $\mathcal{R}(\mathbf{A})$ is the rank of the matrix $\mathbf{A}$.*

This proposition is also presented by Wang and Malakooti [74]. However, it is derived based on an incorrect theorem [66] (see Appendix B). Proposition 3.1 extends results due to Huang and Huang [34] and Arai [4] from threshold to tanh hidden units.

Proposition 3.1 can be extended to multi-output systems. Since a complete basis is guaranteed for one output node by Proposition 3.1, each subsequent output node can form a linear combination of the basis to generate the required target vector. This result is formalized in Corollary 3.2.

Corollary 3.2 *To learn an arbitrary set of N_P* **multi-input multi-output** *patterns with distinct input patterns with an MLP network with direct input to output*

connections,

$$N_P - \mathcal{R}\left(\left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_{N_I} & \mathbf{1} \end{array}\right]\right)$$

tanh *hidden units are required.*

The number of hidden units required in Corollary 3.2 is an upper bound derived based on the worst case scenario. The worst case scenario is based on a network with N_P outputs and where the set of target vectors form a complete basis. In this situation, all the linearly independent activation vectors are needed to span all N_P linearly independent target vectors. Thus all $N_P - \mathcal{R}\left(\left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_{N_I} & \mathbf{1} \end{array}\right]\right)$ hidden units are necessary to insure exact learning. Because of this necessity, the bound derived in Corollary 3.2 is the least upper bound on the number of hidden units required for a network with N_P outputs.

In this investigation, the emphasis has been on producing a complete basis of vectors to guarantee that all targets can be approximated. In describing the ability of the hyperbolic tangent to produce a complete basis, i.e. Corollary 3.1, the choice of vectors to form the complete basis was not unique. In fact, there is a tremendous diversity of linearly independent vectors that the hyperbolic tangent function can produce over the parameters a and b.

Because of the diversity of possible outputs of the hyperbolic tangent it is conceivable that in some instances a complete basis may not be necessary for exact learning.

Consider the problem of a single-input, single-output MLP network with no direct connections from input to output. Note that according to Lemma 3.2 single-input and multi-input networks are equivalent. According to Theorem 3.1, the single-input single-output MLP requires $N_P - 1$ hidden units for exact learning. This method creates a complete basis of linearly independent vectors of the internal representation to guarantee exact learning.

If instead of producing a complete basis, the linearly independent vectors are chosen to span the subspace which the given target vector spans, the number of vectors may be greatly reduced. For example, in the best case, there may be a mapping directly from the input vector through one tanh hidden unit to give the

output vector. The more important bound is the worst case number of hidden units. That is, for some target vector, the diversity of the tanh units, limited by the two parameters a and b, may be *stretched* so that a maximum number of hidden units is required. By Theorem 3.1, the maximum number is $\leq N_P - 1$. This maximum number has been derived by Sontag [68]. Sontag shows that for a one output network a least upper bound on the number of hidden units required for exact learning is $\left\lceil \frac{N_P}{2} \right\rceil + 1$.

This bound corresponds to approximately half the number of required hidden units when compared to the results of Theorem 3.1. The discrepancy lies in the choice of vectors from the diverse hidden unit output created by $\mathbf{tanh}(a\mathbf{u} + b\mathbf{1})$, as opposed to the *brute force* method of constructing a complete basis. That is, by simply producing a set of linearly independent vectors the target being learned is not important; after the basis has been assembled, the target can be learned exactly. However, because of the diversity allowed by the parameters a and b the vectors can be chosen judiciously so that they span only a subspace that covers the target vector. By describing the worst case bound for the single output network, Sontag has shown that the diversity of the hidden unit output can be exploited to surpass simple brute force techniques of generating a complete basis.

Moreover, the Sontag result also shows the limitation of the hidden unit output. The limitation of the hidden unit outputs is as important as the diversity of the output, since it determines the number of hidden units required in the worst case scenario. The important factors in this limitation may be the number of parameters by which the activation function of the unit is defined. With a greater number of parameters in the activation function, the activation function could generate a wider variety of hidden unit vectors. Unfortunately, a single hidden unit, irrespective of the number of parameter, can only be responsible for one vector in the computation of the network. It is shown below that increasing the number of output nodes reduces the relative effectiveness of the hidden unit diversity versus a simple brute force "build a complete basis" method.

The results presented in this section, have all been based on a brute force approach,

i.e. generating a complete basis. Choosing the vectors of internal representation from the diversity of the hidden units outputs can reduce the number of required hidden units as described by Sontag. In comparing the worst cases, the number of required hidden units is approximately halved by using the Sontag result.

However, if the number of output units is increased, the discrepancy between the brute force method and an analogous worst case Sontag results decreases. This is a result of the fact that the brute force method is useful for multi-output systems where there is a large diversity of target vectors. In fact, when the number of output nodes of the network is equal to the number of patterns, the worst case of the analogous Sontag result would result in the same bound as Corollary 3.2.

The study of the diversity and limitations of a hidden unit output will be an important aspect of describing the least upper bounds for networks with number of output nodes between the result of Sontag (i.e. single) and Corollary 3.2 (i.e. N_P output nodes).

The ultimate goal of investigations into the hidden unit diversity is a predictor of the minimal number of hidden units for a particular pattern set. This would result in the optimal architecture for the MLP, thereby minimizing the size and computational requirements of training the network. As yet, an algorithm to determine this *minimal* number of hidden units is an open question.

Linear independence is a necessary property of the internal representation when considering successful training. The results shown indicate that the weights *exist* for exact learning to take place. However, the efficacy of the chosen training algorithm determines if exact learning can be achieved. The next section shows that a common strategy of most training algorithms implicitly increases the number of linear independent vectors in the internal representation, thereby increasing the space of target vectors that can be produced. and hence the likelihood of achieving exact learning.

3.4 Minimizing the Sum of Squared Error Enhances Linear Independence

This section describes how a common strategy for network training implicitly increases the number of linear independent vectors in the internal representation.

Popular strategies in training MLPs formulate training as a nonlinear optimization problem. An objective function representing the performance error of the MLP is minimized over the parameters (i.e. weights) of the MLP. The most common objective function used for MLP training is the sum of squared error. The backpropagation of errors algorithm [62], a widely used training algorithm, minimizes a sum of squared error objective function using a steepest descent algorithm. Other algorithms use other nonlinear optimization techniques or ad hoc modifications thereof [6, 63, 73, 17, 59, 56, 69, 58, 10].

Formally, the sum of squared error function for a one output network is written as

$$\begin{aligned} J &= \frac{1}{2}\sum_{i=1}^{N_P}([\mathbf{T}]_i - [\mathbf{z}]_i)^2 \\ &= \frac{1}{2}\|\mathbf{T} - \mathbf{z}\|^2. \end{aligned}$$

The sum of squared error function can be expanded using the vector form of the MLP computation. This expansion shows that minimization of the sum of squared error implicitly increases the number of linear independent vectors in the internal representation. From Section 3.3 it is clear that with a larger number of linearly independent vectors in the internal representation the network can produce a larger dimension of target vectors and thereby increase the likelihood of achieving exact learning. The derivation presented in this investigation is an extension of a formulation due to Webb and Lowe [75]. Webb and Lowe show that the maximization of a complex nonlinear discriminant function is equivalent to the minimization of the sum of squared error objective function. Using the concepts of linear independence and singular value decomposition, the results of Webb and Lowe can be further simplified

so that the quantities being maximized are related directly to the number of linearly independent vectors in the internal representation and the direction of those vectors with respect to the target vector. This is stated formally in Proposition 3.2.

Proposition 3.2 *Consider an MLP of the form described in Section 3.2, Case III. Let the hidden to output and input to output weights,* $\mathbf{W}_\Omega$*, minimize the sum of squared error at all times, as prescribed by Webb and Lowe [75]. Let* $J^\star$ *be equal to* J *with* $\mathbf{W}_\Omega$ *chosen to minimize the sum of squared error at all times. Then,*

$$\min_{\substack{\mathbf{W}_{\Lambda_i} \\ i=1,\ldots,N_H}} J^\star \tag{3.12}$$

maximizes the number of linearly independent vectors in the internal representation.

Proof:

Substituting from equation 3.11,

$$J = \frac{1}{2}\|\mathbf{T} - \mathbf{Y}\mathbf{W}_\Omega\|^2 \tag{3.13}$$

where $\mathbf{Y}$ is the matrix formed by the column vectors in the set of activation vectors, $\{\mathbf{y}_1, \ldots, \mathbf{y}_{N_H}, \mathbf{v}_1, \ldots, \mathbf{v}_{N_I}, \mathbf{1}\}$.

As prescribed in Webb and Lowe [75], let $\mathbf{W}_\Omega$ be the least squares solution of $\mathbf{T} = \mathbf{Y}\mathbf{W}_\Omega + \mathbf{e}$, thereby eliminating the dependence of the cost, J, on $\mathbf{W}_\Omega$. The general solution to the least square problem is given by:

$$\mathbf{W}_\Omega = \mathbf{Y}^\dagger\mathbf{T} - (\mathbf{I} - \mathbf{Y}^\dagger\mathbf{Y})\mathbf{u} \tag{3.14}$$

where $\mathbf{Y}^\dagger$ is the Moore-Penrose inverse and $\mathbf{u}$ is an arbitrary vector [9]. This family represents the most general set of solutions, often the solution, $\mathbf{W}_\Omega = \mathbf{Y}^\dagger\mathbf{T}$, is chosen in practice. Equation 3.13 can be expanded to :

$$J = \frac{1}{2}(\mathbf{T}^T\mathbf{T} - 2\mathbf{T}^T\mathbf{Y}\mathbf{W}_\Omega + \mathbf{W}_\Omega^T\mathbf{Y}^T\mathbf{Y}\mathbf{W}_\Omega)$$

subsequent substitution of Equation 3.14 gives :

$$J^\star = \frac{1}{2}(\mathbf{T}^T\mathbf{T} - 2\mathbf{T}^T\mathbf{Y}(\mathbf{Y}^\dagger\mathbf{T} - (\mathbf{I} - \mathbf{Y}^\dagger\mathbf{Y})\mathbf{u}) + (\mathbf{Y}^\dagger\mathbf{T} - (\mathbf{I} - \mathbf{Y}^\dagger\mathbf{Y})\mathbf{u})^T\mathbf{Y}^T\mathbf{Y}(\mathbf{Y}^\dagger\mathbf{T} - (\mathbf{I} - \mathbf{Y}^\dagger\mathbf{Y})\mathbf{u})) \quad (3.15)$$

It is important to note that $\mathbf{Y}^\dagger$ satisfies the following Moore-Penrose equations [9].

$$\mathbf{Y}\mathbf{Y}^\dagger\mathbf{Y} = \mathbf{Y} \quad (3.16)$$

$$\mathbf{Y}^\dagger\mathbf{Y}\mathbf{Y}^\dagger = \mathbf{Y}^\dagger \quad (3.17)$$

$$(\mathbf{Y}^\dagger\mathbf{Y})^T = \mathbf{Y}^\dagger\mathbf{Y} \quad (3.18)$$

$$(\mathbf{Y}\mathbf{Y}^\dagger)^T = \mathbf{Y}\mathbf{Y}^\dagger \quad (3.19)$$

Expanding equation 3.15 with generous use of equations 3.16 through 3.19 gives,

$$J^\star = \frac{1}{2}(\mathbf{T}^T\mathbf{T} - 2\mathbf{T}^T\mathbf{Y}\mathbf{Y}^\dagger\mathbf{T} + \mathbf{T}^T\mathbf{Y}^{\dagger T}\mathbf{Y}^T\mathbf{Y}\mathbf{Y}^\dagger\mathbf{T})$$

which can be simplified by using the Moore-Penrose equations as :

$$J^\star = \frac{1}{2}\mathbf{T}^T(\mathbf{I} - \mathbf{Y}\mathbf{Y}^\dagger)\mathbf{T} \quad (3.20)$$

Minimizing $J^\star$ over the remaining free weights (i.e. the input to hidden weights) is equivalent to:

$$\arg \min_{\substack{\mathbf{W}_{\Lambda_i} \\ i=1,\ldots,N_H}} J^\star = \arg \max_{\substack{\mathbf{W}_{\Lambda_i} \\ i=1,\ldots,N_H}} \mathbf{T}^T\mathbf{Y}\mathbf{Y}^\dagger\mathbf{T} \quad (3.21)$$

A singular value decomposition [9] yields:

$$\mathbf{Y} = \mathbf{U}\Gamma\mathbf{V}^T \quad (3.22)$$

$$\mathbf{Y}^\dagger = \mathbf{V}\Gamma^\dagger\mathbf{U}^T \quad (3.23)$$

$$\Gamma\Gamma^{\dagger} = \left[\begin{array}{c|c} \mathbf{I}_{\bar{r}\times\bar{r}} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array}\right] \tag{3.24}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices, Γ is a diagonal matrix and $\bar{r}$ is the number of linearly independent columns of $\mathbf{Y}$. The assumption has been made that the first $\bar{r}$ columns of $\mathbf{Y}$ are linearly independent of each other, to simplify the exposition. This assumption can be made without loss of generality. Moreover, the first $\bar{r}$ columns of $\mathbf{U}$ form an orthonormal basis for the space spanned by the linearly independent columns of $\mathbf{Y}$ [14]. Thus the first $\bar{r}$ of $\mathbf{U}$ form an orthonormal basis for the output space of the network.

Using equations 3.22 to 3.24, the right-hand side equation 3.21 can be written as :

$$\arg \max_{\substack{\mathbf{W}_{\Lambda_i} \\ i=1,\ldots,N_H}} \mathbf{T}^T\mathbf{U} \left[\begin{array}{c|c} \mathbf{I}_{\bar{r}\times\bar{r}} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} \end{array}\right] \mathbf{U}^T\mathbf{T} \tag{3.25}$$

Expanding Equation 3.25 results in :

$$\arg \max_{\substack{\mathbf{W}_{\Lambda_i} \\ i=1,\ldots,N_H}} \sum_{j=1}^{\bar{r}} (\mathbf{T}^T\mathbf{u}_j)^2 \tag{3.26}$$

where $\mathbf{u}_j$ is the j^{th} column of $\mathbf{U}$.

In studying the summation in Equation 3.26, the parameters which influence the maximization are directions of the columns of $\mathbf{U}$ and the number of linearly independent vectors in the columns of $\mathbf{Y}$, $\bar{r}$.

The first $\bar{r}$ columns of $\mathbf{U}$ span the output space as a result of the singular value decomposition of Equation 3.22. Thus $\mathbf{T}^T\mathbf{u}_j$ represents the projection of the target vector onto an orthonormal basis of the output space. That is, the maximization term in Equation 3.26 represents the norm of the projection of the target vector onto the output space. Effectively, this measures *how much* of the target has been spanned by the columns of $\mathbf{Y}$. When this norm in Equation 3.26 is equal to the norm of

$\mathbf{T}$ in $\Re^{N_P}$ then the network has learned the target exactly.

In the worst case when a complete basis of vectors forms from the internal representation, $\bar{r} = N_P$, and the number of linearly independent vectors of the internal representation would be $N_P - \mathcal{R}\left(\left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_{N_I} & \mathbf{1} \end{array}\right]\right)$ as described in Proposition 3.1.

In the worst case scenario, the most important quantity in Equation 3.26 is $\bar{r}$. Without the required number of linearly independent vectors, the target cannot be learned exactly. In fact, if the target is learned via smaller number of linearly independent vectors (i.e. not worst case), this may indicate an inherent redundancy in the input-output patterns. As a result, some of the patterns can be eliminated and the resulting set of patterns will have this worst case requirement of linear independence.

Via the equivalence of Equation 3.26 and Equation 3.21, the minimization of the sum of squared error with $\mathbf{W}_\Omega$ chosen as a least squares solution increases the *number* of linearly independent vectors in the internal representation to achieve exact learning.

□

The above proof, although rather technical, uses simple concepts to reveal the relationship between the internal representation and the sum of squared error. The concept of a pseudoinverse is used to eliminate the dependence of the error function on the output to hidden weights, $\mathbf{W}_\Omega$. These weights, although important for the calculation of the output (Equation 3.10), do not affect the internal representation (Equations 3.4 and 3.5). In fact, these weights are chosen to minimize the squared error *based on* the internal representation. The result of the pseudoinverse analysis is an equivalent objective function which depends only on the internal representation and the target vector.

To further simplify the objective function, a singular value decomposition (SVD) is used. This technique allows a matrix to be expressed as the product of orthonormal and diagonal matrices. The number of non-zero elements in the diagonal matrix is

the number of linear independent columns of the original matrix. Using an elegant relationship between the SVD of a matrix and the SVD of its pseudoinverse, the minimization of the objective function can be manipulated into the maximization of the weighted norm of the target vector. The weights for the norm are dependent on both the direction and number of the linearly independent vectors in the internal representation. However, if the worst case scenario is considered, that is all the linearly independent vectors are required, then the direction of the linearly independent vector becomes immaterial. Exact learning cannot take place without all the vectors present, thus the number of linearly independent vectors is the more important maximization than the directions of the linearly independent vectors.

Thus, the minimization of the sum of squared error objective function (with least squares error $\mathbf{W}_{\Omega}$) maximizes the number of linear independent vectors in the internal representation.

As seen in the previous section, a larger number of linearly independent vectors in the internal representation also increases the dimension of the target space that is representable. Thus, reduction of the sum of squared error increases the dimension of target vectors that can be produced and tunes those vectors to span the subspace spanned by the target vector. In the context of training, the increase in dimension increases the likelihood that the MLP will learn the required target vector exactly.

3.5 Discussion

This investigation examines the role of a linearly independent internal representation in the successful training of sigmoidal MLPs. Although results exist for threshold MLPs, sigmoidal activation functions are widely used because of the increased storage capacity and differentiability. Ironically, sigmoidal MLPs pose a difficult problem in training analysis because of the continuous nature of their activation functions.

By manipulating the internal representation into a node-wise vector form, the computation of the MLP output can be chosen to be a linear combination of the vectors of the internal representation and the input vectors (Equation 3.10). As such,

the linear independence of vectors of the internal representation with respect to the input vectors increases the dimension of possible outputs of the MLP. The increase in dimension of possible output increases the likelihood of achieving exact learning. The results in this investigation suggest the important roles of the activation function, architecture and training algorithm in exact learning through the linear independence of the internal representation.

First, the sigmoidal activation function has the *ability* to produce the required linearly independent vectors (see Lemma 3.1). This result is analogous to statistical results due to Oh and Lee [53]. The ability of producing linearly independent vectors is a fundamental result in describing their role in exact learning.

Secondly, the *architecture* must allow for the required number of sigmoidal hidden units to be present. Without enough hidden units, the number of linearly independent vectors that can be produced will be limited thereby limiting the dimension of the output space of the MLP.

To guarantee exact learning of an arbitrary set of patterns, a complete basis of vectors must contribute to the computation the output. A complete basis will be able to generate any possible target vector. Proposition 3.1 defines the minimum number of hidden units required to guarantee a complete basis and therefore exact learning. Proposition 3.1 extends the results due to Huang and Huang [34] and Arai [4] from threshold units to sigmoidal units.

Work by Sontag [68] has shown that for a *given* set of single-output patterns the least upper bound on the number of hidden units is smaller than the minimum number for an arbitrary set (Proposition 3.1). This follows from the fact that the vectors of the internal representation can be modulated through the diverse output of the tanh function to span the subspace spanned by the single target vector rather that creating a complete basis. The properties of the tanh function allow enough diversity, as shown by Sontag, to require about half the number of units for a single output network in the worst case, when compared to Theorem 3.2. For multiple output networks, the goal of specificity, seen in the Sontag result and the goal of a complete basis, are combined to ensure exact learning. Moreover, the discrepancy between the number of worst case

hidden units required for the Sontag approach versus the complete basis decreases with the increasing number of output units. When the number of outputs equal to the number of pattern the worst case of the Sontag method and the complete basis method intersect — the activation vectors must form a complete basis (Corollary 3.2) irrespective of the method used in choosing the vectors.

Computation of the *minimum* number of hidden units for a *given* set of input-output patterns would produce an optimal architecture. Unfortunately, such results do not exist. The possible role of linear independence and hidden unit diversity in developing results for optimal architecture prediction is, as yet, unclear.

Having established both the *ability* and *architecture* for the linearly independent internal representation, the *production* of these vectors is the last requirement for exact learning. The production of a linearly independent internal representation is shown to be a direct result of training. The minimization of the sum of squared error, a common training strategy, implicitly increases the number of linearly independent vectors in the internal representation. This relationship was shown by extending a result due to Webb and Lowe [75]. The practical fallout of this results is that common training algorithms such as backpropagation of errors [62] implicitly increase the linear independence in an MLP network thereby increasing the likelihood of exact learning. Work by Orfanidis [55] *explicitly* increases the linear independence of the hidden unit outputs by using a Gram-Schmidt processor. The result is accelerated convergence, as expected.

The linear independence of the internal representation is a necessary property for exact learning. This feature of learning may be a potentially useful substrate for the design of new algorithms. Further investigations in this area will hopefully show that linear independence can be used effectively to increase neural network training performance and reduce the computational burden currently required to train MLPs.

Chapter 4

Tree Structured Neural Network Architectures - The Tree-Like Perceptron Network

4.1 Introduction

The design of neural network architectures has, historically, been divided into two strategies. The engineering perspective prescribes that neural network architectures must be effective and computationally efficient. Designs range from radically new computational structures [31, 1, 18] to reformulation of existing architectures to make more efficient use of computational resources such as VLSI implementations or parallel computers [44, 78, 27, 45, 51, 37, 18, 46, 35, 50]. The hope is that such designs will yield strategies to learn faster and to a high degree of accuracy.

Another strategy in designing neural network architectures is biological relevance. That is, designing architectures that are faithful to the computation and organization of the brain. The working assumption in this biological strategy is that the brain functions in a manner that can be modeled on existing computational structures. The earliest models of artificial neural networks, such as the single layer perceptron, have taken inspiration from the human brain. And now as the field of neurobiology

is uncovering important mechanisms in the learning process, new models have even a greater advantage in addressing the need for biological plausibility.

These design strategies are not mutually exclusive. The efficiency and efficacy of the human brain for learning is a paradigm for the design of biologically relevant and computationally efficient architectures. Moreover, as the size of artificial neural network applications increases and surpasses the available computational resources, this dual strategy becomes increasingly attractive.

In this investigation, spanning this chapter and the next, a novel family of neural network architectures, the Tree Structured architectures, is described. The design of this family of architectures uses both biological evidence and analysis of existing training strategies to create a highly parallelizable learning model.

This investigation describes two Tree Structured architectures derived from the Multilayer Perceptron: The Tree-Like Perceptron network described in this chapter and the Tree-Like Shared network described in Chapter 5. The motivation and implementation details of each architecture are described and simulations demonstrate their performance when compared to the conventional Multilayer Perceptron (MLP).

4.2 The Tree Structured Architectures

The Tree Structured architectures are a family of modularized neural network models based on a *divide and conquer* approach to training. That is, a task is partitioned into a group of subtasks and each subtask is solved individually.

For example, consider a multi-output MLP with one hidden layer (one hidden layer is sufficient for any continuous task [33]). Using the decomposition strategy, the MLP network is subdivided into a set of single output subnetworks. Each single output subnetwork has one hidden layer, independent of other subnetworks; and the same number of inputs as the original MLP. Each subnetwork is trained with one element of all the output patterns, and all the input patterns. That is, each subnetwork learns the relationship between the input, and one feature of the output. Immediate benefits include the possibility of parallel training and the subdivision of application size.

The *divide and conquer* approach forms the basis of a group of neural network models called modular networks [31, Chapter 12]. Modular networks represent a class of networks in which a large task is decomposed into a set of smaller subsystems in some application dependent fashion. For example, work by Rueckl et al [61] shows how a single MLP network used for shape and location identification can be improved by decomposing the single MLP network into two MLP subnetworks, one responsible for the shape task and the other responsible for the location task. This type of modularization strategy is useful when the task can be decomposed along functional relationships.

The Tree-Like Architectures differ from conventional modular networks in that each output of the system, irrespective of its function relative to other outputs, is decomposed into a module. As will be shown, this type of modularization has computational advantages. In addition, the Tree-Like decomposition strategy is not application dependent and thus can be used even when a functional decomposition is not obvious.

4.2.1 Motivation

This section describes some of the motivations, both biological and computational, for the decomposition strategy of the Tree Structured architecture.

The Credit Assignment Problem in the MLP

Credit assignment [31, 47] refers to the allocation of blame within a learning representation. Each parameter in a learning representation is updated according to some error signal indicating that the learning process has not reached an adequate level. Credit assignment can become a problem when a parameter must respond to several error signals simultaneously. In this case, the error signals can all be non-zero, but the combination of these signals may result in no change in the parameter. That is, the learning is inadequate, but the parameters are not changing. This results in suboptimal learning.

In a multi-output MLP, there is an error signal for each output of the network. For the hidden to output weights, the credit assignment is simple, each of these weights contributes to only one output. However, the input to hidden weights contribute to all the outputs of the network. The result is that the input to hidden weights are updated according to some combination of the error signals [62]. Thus each input to hidden weight can receive conflicting update instructions from each output, this is the credit assignment problem. It is unclear, as yet, how credit assignment may manifest itself in learning. If parameter updates do indeed stop even when error is non-zero, it may indicate a relationship between credit assignment and local minima, however this has never been shown.

The Tree Structured approach to the MLP eliminates this credit assignment problem by allocating a subnetwork to each output. All the weights in a particular subnetwork contribute to the production of one output. As a result, each weight in the network is updated by only one error signal, thereby eliminating the possibility of multiple error signals.

Parallel Computational Substructures

The parallelization induced by the decomposition process has many important features.

First, the modularity of each subtask allows partial training of the entire task. That is, a portion of the task can be learned, thereby allowing the available computational resources to be focused on the remaining subtasks. This feature may not be possible in the undecomposed larger task. For example, training an MLP network is either adequate or not, no partial information can be extracted from a particular training session.

The decomposition strategy also allows for parallel training. Instead of training a large task which is computationally intensive, a set of subtasks, each with smaller computational requirements, can be trained simultaneously. In general, this reduces the required training time of the parallel subtasks over the original task. In addition, each subnetwork is independent, thus, tightly coupled architectures described

in parallelization schemes such as Zhang [78] are not required, each subnetwork can be trained in a distributed fashion without any intersubnetwork communication.

In principle, the decomposition process requires no communication between subnetworks. However, Chapter 5 describes how restricted communication between the subnetworks may enhance training and reduce required computational resources.

Decomposition can also result in robust performance of the working implementation of the learned task. Once training is complete, the subtasks act in parallel to perform the specified task. Because of the modular structure, each subtask can be duplicated adding redundancy to the implementation. Such redundancy is possible even when decomposition is not used, however, the implementation for parallel, independent subtasks is substantially simpler.

Modular Processing in the Human Brain

There is increasing evidence that information processing in the brain is performed by parallel subsystems [30, 77]. That is, systems such as vision and language may be composed of subsystems that act on input information (e.g. retinal image or audible speech) and process it in parallel. This type of information processing paradigm has been fundamental in the development of artificial neural networks.

Recent evidence indicates that parallel learning may also take place in the brain [41]. In the same way that parallel subsystems are involved in the processing of information, there may exist parallel subsystems that store information. This represents an important paradigm shift for artificial neural networks. Although parallel learning is known to be computationally efficient, there is now evidence that it may also be biologically plausible.

In this way, the Tree Structured architectures represent both a computationally efficient and biologically relevant model of learning.

4.3 Tree-Like Perceptron Network

This section describes the simplest form of the Tree Structured architecture, the Tree-Like Perceptron network.

4.3.1 Architecture

The architecture of the Tree-Like Perceptron network is simplest application of decomposition strategy. The multi-output MLP is partitioned into a set of single output subnetworks, one for each output in the original network. A simple example of the decomposition process is shown in Figure 4-1.

The allocation of hidden units to each subnetwork is arbitrary. If the original MLP network has a specified number of hidden units, then a simple strategy for allocation is to distribute the hidden units equally amongst the subnetworks. Otherwise, the number of hidden units can be chosen initially and then changed based on the training behavior, a common strategy for MLP networks.

An important fact of hidden unit allocation is that the number of hidden units in any subnetwork is guaranteed to be no larger than the total number of hidden units to train the MLP network. If this fact were not true, it is possible for a subnetwork to require more computation to train than the original task. This fortunate result can be demonstrated by using the concept of linear independence developed in Chapter 3.

If a multi-output MLP requires a minimum of M hidden units to be trained, then there are M linearly independent vectors in the internal representation. That is, if any of the vectors of the internal representation were linearly dependent, then the associated hidden unit would be unnecessary. Moreover, the target vector of each output lies in the space spanned by the M linearly independent vectors. Thus any target output of a subnetwork must also lie in a space spanned by the M linearly independent vectors. By placing the M linearly independent vectors into the internal representation of the subnetwork, the subnetwork can now learn its target vector. In fact, the number of hidden units required by the subnetwork may be much smaller than M.

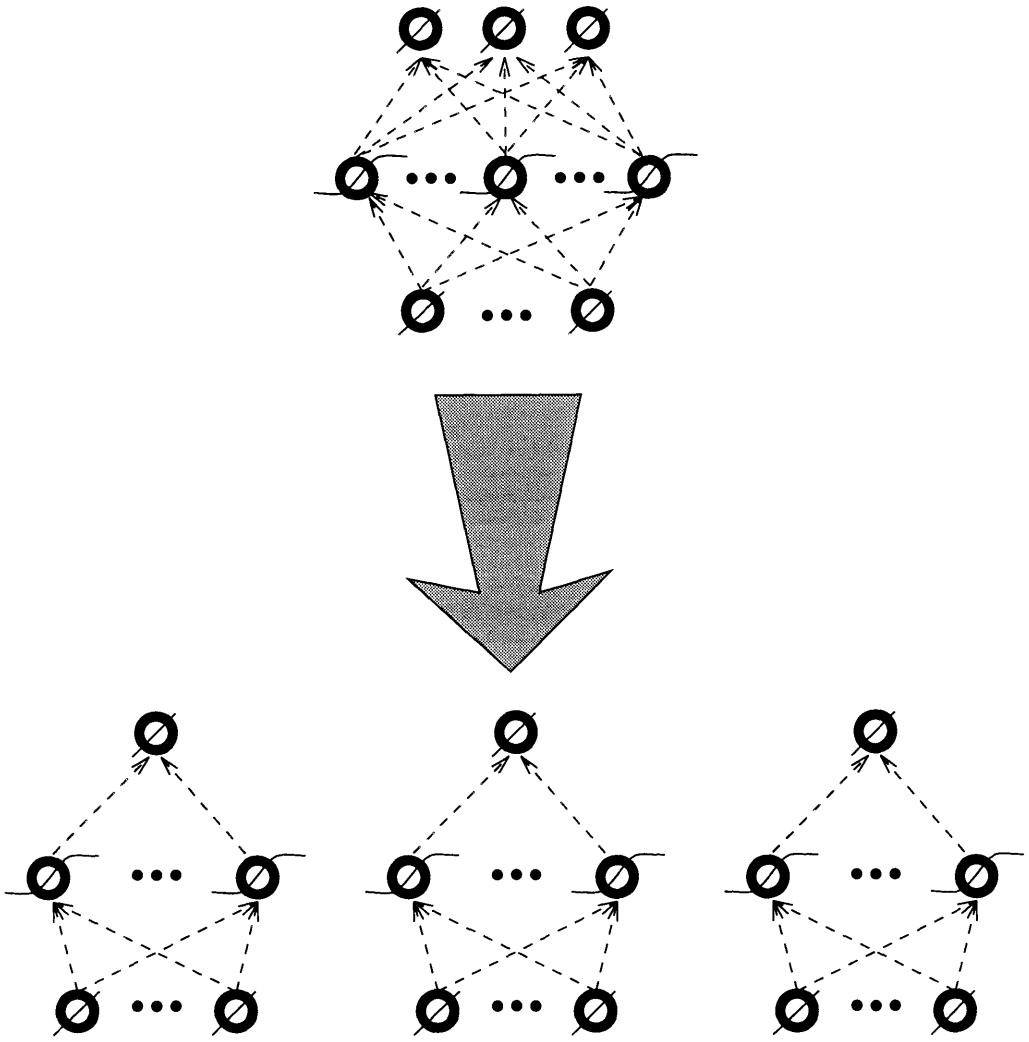


Figure 4-1: Decomposition of MLP into a Tree-Like Perceptron network

Having established the architecture of the subnetworks, the issue of training becomes important. The next section outlines a measure for gauging the utility of the Tree-Like Perceptron network over the MLP for particular applications. In addition, some novel strategies are discussed that take advantage of the decomposition approach, even in computationally-limited environments.

4.3.2 Training

In evaluating Tree-Like Perceptron training, the MLP network was chosen as the architecture for comparison. This choice was made due to both the availability of MLP training software and its wide use in artificial neural network applications [28].

One method of evaluating large MLP performance is to model the training process probabilistically. In training large MLP networks, training sessions are run repeatedly until a desired level of training is reached. Each training session can be treated as a Bernoulli trial. That is, over each training session the MLP network either learns adequately with some probability p or does not learn adequately with probability $1 - p$. Using this approach, the expected training performance can be calculated through two parameters : the probability of learning adequately on any particular session, p, and the time required to run the training session, T_S. If these quantities are known, the expected amount of time required to train the MLP is $\frac{T_S}{p}$.

Using a similar training approach for the Tree-Like Perceptron network a relative performance measure can be calculated. Training performance of the Tree-Like Perceptron, however, will vary widely with the available computational resources. If extensive computational resources are available, then a fully parallel training scheme will result in the minimum training time. With smaller resources, partial parallel or even serial approaches may be required.

Serial Subnetwork Training

Assume that the subnetworks have to be trained one at a time due to lack of computational resources, this is the worst case training performance of the Tree-Like

Perceptron network.

Using a protocol similar to that of the MLP network, each subnetwork can be trained for one session and then evaluated. If the subnetwork learns to an adequate level, then the parameters are stored and this subnetwork no longer needs to be trained again. However, if the subnetwork does not learn to an adequate level, then the subnetwork is put into a queue to get retrained. The training continues with the next subnetwork. This retraining of subnetworks continues until all subnetworks have been trained to an adequate level. Given this training protocol, the serial training performance of Tree-Like Perceptron can be calculated.

Let a session of MLP network training be T_{MLP} units of time in duration. Let the probability of the MLP converging (i.e. reaching an adequate level of training) on any training session be P_{MLP}. Due to the smaller size of the Tree-Like Perceptron network subnetwork, a training session of the subnetwork will be a fraction of the time that a MLP takes to train say, $\beta_{TLP} T_{MLP}$, where $\beta < 1$. If the probability of the convergence of any Tree-Like perceptron subnetwork is P_{TLP} (assume for simplicity that each subnetwork is equally likely to converge) and there are N_o output units, then

$$\begin{aligned} C_{MLP} &= T_{MLP} \frac{1}{P_{MLP}} \\ C_{TLP} &= \beta_{TLP}\, T_{MLP} \frac{N_o}{P_{TLP}} \end{aligned}$$

where C_{MLP} and C_{TLP} are the computation times for the MLP and the Tree-Like Perceptron network respectively. The ratio of these computation times, $\frac{C_{MLP}}{C_{TLP}}$ is a measure of the efficiency of the Tree-Like perceptron network performance:

$$\frac{C_{MLP}}{C_{TLP}} = \frac{P_{TLP}}{P_{MLP}} \frac{1}{\beta_{TLP}\, N_o}.$$

If the ratio $\frac{C_{MLP}}{C_{TLP}} > 1$ then the Tree-Like Perceptron network trains in a shorter amount of time, otherwise the MLP converges faster. This measure cannot be directly computed since the probabilities of convergence are unknown. However, the ratio of

probabilities required to choose the Tree-Like Perceptron, the critical ratio, $\theta^{\star}_{serial}$, can be computed as :

$$\frac{P_{TLP}}{P_{MLP}} > \beta_{TLP}\, N_o = \theta^{\star}_{serial} \tag{4.1}$$

That is, for the serially trained Tree-Like Perceptron network to outperform the MLP, the ratio of probabilities must be greater than $\theta^{\star}_{serial}$, the critical ratio for the serial training. This measure indicates how much *easier* the Tree-Like Perceptron subnetworks must be to train (i.e. greater probability of convergence) for the Tree-Like Perceptron to be more effective than the MLP network at learning the set of patterns. This measure can also be adapted to parallel training schemes, where the effectiveness of the decomposition is greater.

Parallel Subnetwork Training

If the computing resources are increased so that N subnetworks, $N \leq N_o$, can be trained simultaneously, then N subnetworks are trained in parallel until all subnetworks reach an adequate level of learning.

Following the serial training derivation, the critical ratio for N-parallel training, $\theta^{\star}_{N}$, is :

$$\theta^{\star}_{N} = \beta_{TLP}\, \frac{N_o}{N}. \tag{4.2}$$

This measure, when compared to the serial critical ratio, is much smaller due to the presence of N in the denominator. In fact, as N increases toward N_o, the ratio can become less than one, indicating that even if the single output network has a lower probability of convergence than the MLP network, the speed of each training session and parallel nature of training make the Tree-Like Perceptron network more attractive computationally, than the MLP.

Simultaneous Subnetwork Training Sessions

Consider the case where the number of subnetworks to be trained is *smaller* than the maximum number of subnetworks that can be trained in parallel. This situation can arise in two different ways: the computational resources may be great enough

to support more than the N_o subnetworks resulting from the decomposition; or the number of subnetworks to be retrained falls below the maximum number during the parallel training process. In either case, the extra parallel *slots* can be used to train the same subnetwork in multiple sessions simultaneously. In this way, the computational resources are maximized and the probability of convergence for the subnetwork is increased.

Adaptive Resource Allocation

In considering the training of the Tree-Like Perceptron, the assumption has been that the original choice of hidden layer size for each subnetwork was sufficient. In practice, some of the subnetworks may not converge after many retraining sessions, i.e. $P_{TLP} \approx 0$. One method of solving this problem is to dynamically change the size of the hidden layers in those subnetworks which are consistently failing to achieve an adequate level of training. It is known that increasing the size of the hidden layer increases the possible number of linearly independent vectors in the internal representation (Section 3.2). This also increases the likelihood of achieving exact learning, thereby increasing P_{TLP}. Rueckl et al [61] describe a similar technique used to develop efficient internal representations in split MLP networks, using a form of decomposition similar to the paradigm of modular neural network models.

Hidden Credit Scaling

One possible problem with the decomposition process is related to credit assignment. Although the credit assignment *problem* is eliminated in the Tree-Like Perceptron network, the amount of credit each input to hidden weight sees in a subnetwork decreases. In a multi-output MLP, input to hidden weights respond to a combination of the error signals at the output. Although the error signals could negate each other, they may also reinforce each other causing large parameter changes. In contrast, the single output of the subnetworks provides a fraction of the error signal that an input to hidden weight in an MLP may see resulting smaller parameter changes and slower convergence. To overcome this problem, the credit to each input to hidden weight

can be scaled to increase the response of the weight. This type of approach has also been effective in the training of MLP networks [2].

4.4 Simulation Results and Discussion

This section discusses the results of training simulations of the Tree-Like Perceptron network versus the MLP. Results are given for two different training tasks.

4.4.1 Computer Speech

Experimental Protocol

This simulation example encompasses a Computer Speech task and is discussed in detail in Appendix C. For the purposes of this discussion, the example has approximately 5400 input-output patterns. There are 203 inputs and 26 outputs, all binary valued.

The MLP was chosen to have 100 hidden units. Each subnetwork of the Tree-Like Perceptron was also chosen to have 100 hidden units each to take advantage of the available resources. All the training sessions were run on the Adaptive Solutions CNAPS Neurocomputer (Appendix D).

The simulations were normalized for time. That is, each architecture was allotted a specified amount of time to run as many training sessions as possible. Each architecture was given 60 hours and each training session was limited to 1000 epochs. An epoch is a presentation of all the patterns of the training data once.

The best network for the MLP architecture was chosen based on lowest classification error. The best network for the Tree-Like Perceptron architecture was comprised of the subnetworks with the lowest classification errors over all the training sessions. Classification error is the percentage of incorrectly classified input patterns with respect to the entire training set.

It is important to note that the classification error calculated for the Tree-Like Architecture is a worst case calculation. Classification error is the number of incor-

rectly classified input patterns. Thus, if one of the output nodes does not match the target, the entire pattern is considered erroneous. When decomposing the network into subnetworks a classification error is calculated for each subnetwork. Since it is unclear from which patterns these errors came, the errors can not be consolidated on a per pattern basis. Thus the classification errors are simply added together over all the subnetworks. This number then represents the situation where each error from each subnetwork is from a different example — the worst case. If some of the errors were, in fact, from the same patterns, then in principle, the true classification error would be smaller.

The training method used was backpropagation of errors with a learning rate of 0.01 and no momentum. No adaptive allocation scheme was used for the Tree-Like Perceptron.

Simulation Results and Discussion

In the 60 hours allocated, 100 training sessions of the MLP were run. Nine training sessions of the Tree-Like were run which included 126 subnetworks. Figure 4-2 shows the number of subnetworks trained in each training session. After the last training session 12 subnetworks out of the 26 had reached perfect classification.

A note regarding the use of the critical ratio should be made. The critical ratio, in principle, has the predictive power to decide whether the Tree-Like Perceptron is suited to a particular application. However, the development of the critical ratio assumed that the probabilities of convergence of the subnetworks were the same, given Figure 4-2, this assumption is probably false. Because of the severe limitation in both approximating probabilities of convergence, and modeling multiple probabilities, the critical ratio in unlikely to have any consistent predictive value. At this stage the critical ratio serves to demonstrate the relationships between the important quantities in the training process : the probabilities of convergence (P_{MLP}, P_{TLP}), the relative speed of a training session (β_{TLP}), the number of subnetworks (N_o) and the degree of parallelism (N). Further investigations may reveal new relationships which will allow the critical ratio, or some other measure, to become an accurate predictor of

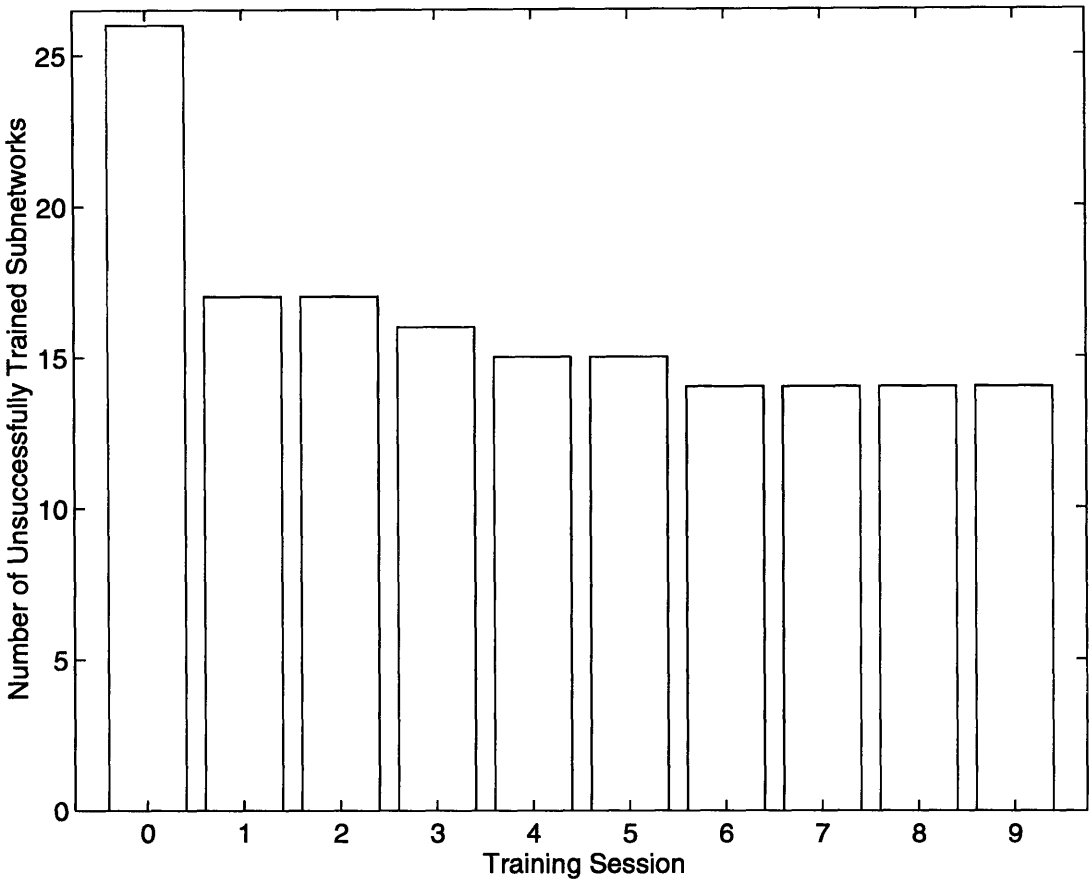


Figure 4-2: Remaining Tree-Like Perceptron subnetworks which have non-zero classification error after each training session. This simulation represents the Computer Speech task.

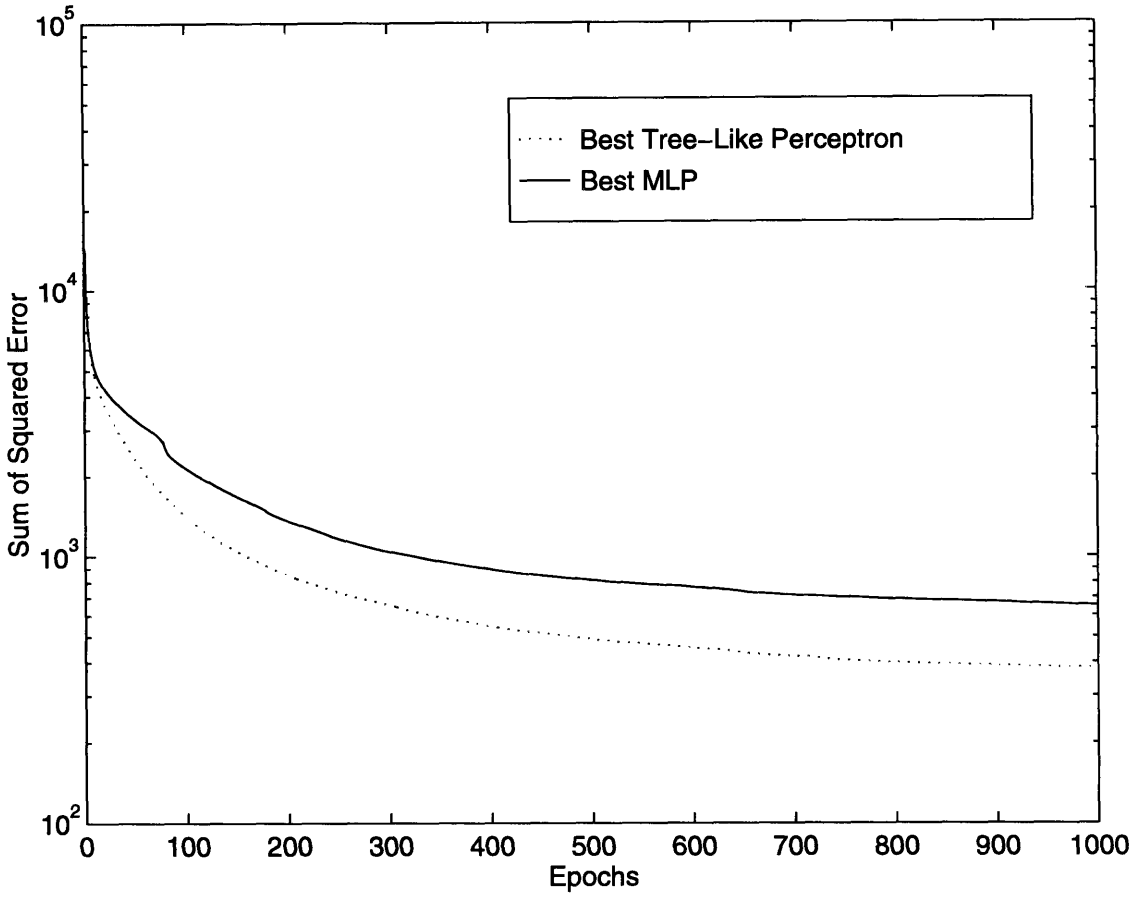


Figure 4-3: Error Trajectories for MLP versus Tree-Like Perceptron for the Computer Speech. The *best* network was chosen according to the description in the text.

Tree-Like Perceptron efficacy.

Because of the difficulty in computation of the critical ratio, the identification of the most efficient architecture is left to simulation.

The error trajectories of the Tree-Like Perceptron and MLP are given in Figure 4-3. The final sum of squared errors are the same order of magnitude, however, comparison of the classification error shows the Tree-Like Perceptron to have greater efficacy. The best classification errors are given in Table 4.1. Neither of the architectures achieved exact learning. However, the Tree-Like Perceptron achieved a classification error of 5.92% whereas the best MLP training session was 12.52%

Note that even after 100 training sessions, the MLP was unable to converge. However, after only 9 training sessions, the Tree-Like Perceptron Architecture was

Architecture	Best Classification Error	
	Number of Errors	Percentage
MLP	681	12.52 %
Tree-Like Perceptron	322	5.92 %

Table 4.1: Best Classification Errors – Computer Speech. The classification errors for each network is given. The Tree-Like Perceptron classification is a worst case calculation.

able to decrease the classification error from the best MLP training session by $\approx 53\%$.

In this example, the MLP network is unlikely to learn exactly due to the limited number of hidden units. As such, this example demonstrates the use of the Tree-Like Perceptron in a resource limited environment. This benefit is a result of the partial training characteristics. For applications which are larger that the available resources, the problem can be decomposed into subproblems which may be solvable in the available resources. In this example, the large MLP problem could not be solved with 100 hidden units (the hypothetical maximum), but the classification error was cut in half by using subnetworks each with 100 hidden units. The unfortunate result of this method is the increase in storage requirements due the large number of parameters.

As described, this Computer Speech problem, cannot be solved in the available resources. It does demonstrate the utility of partial learning, however, the Tree-Like Perceptron, may be unable to achieve exact learning. The example described in the next set of simulations is one in which exact learning can be achieved for the MLP and thus represents another important test of the decomposition strategy.

4.4.2 Character Recognition

Experimental Protocol

The Character Recognition example used in this simulation is discussed at length in Appendix C. For the purposes of this discussion, the set of patterns contains 2600 input-output patterns. There are 64 inputs and 26 outputs, all binary valued.

The number of hidden units in the MLP was chosen to be maximum. Since these

simulations were run on an Adaptive Solutions CNAPS Neurocomputer [27] with 128 processing elements, the maximum size of the MLP network was 500 hidden units. For a detailed discussion of the Adaptive Solutions CNAPS Neurocomputer see Appendix D. The allocation of Tree-Like Perceptron hidden units was 20 hidden units per subnetwork, approximately 500 hidden units among 26 subnetworks. This approach reduces the storage requirements for weights when compared to the previous example.

The simulations were normalized with respect to time. That is, each architecture was allotted a specified amount of time to run as many sessions as possible. Each session was also limited to 10000 epochs. The time allotted to each architecture was 80 hours. At the end of the allotted time, the *best* set of network parameters from each architecture was selected as the basis for comparison.

The *best* set of network parameters was chosen based on two criteria, the final sum of squared error and classification error.

The construction of the *best* set of network parameters differed for each architecture. For the MLP, the best set of parameters resulted from the training session with the lowest sum of squared error. If another training session resulted in fewer *classification* errors, then these parameters were used for the classification performance.

The best Tree-Like Perceptron network was constructed of the best training session for each subnetwork. The best training session for a subnetwork was chosen on the basis of fewest classification errors or sum of squared error. It should be noted that if a subnetwork learned to perfection, i.e. had zero classification errors, the subnetwork was no longer trained, thereby increasing the number of sessions available to other subnetworks.

The training method used for both architectures was backpropagation of errors with a learning rate of 0.01 and no momentum.

This example was chosen because the MLP architecture can learn to perfection, however, the MLP architecture requires a hidden credit scaling factor of 8.0. Since the hidden credit scaling factor is limited to 16.0 on the Adaptive Solutions CNAPS Neurocomputer, it is inappropriate to run the scaled MLP against the Tree-Like Per-

ceptron, since the Tree-Like Perceptron would be unable to generate the hidden credit scale required to compensate for the smaller hidden credit. An appropriate comparison is to run the MLP without hidden credit scaling and the Tree-Like Perceptron with a maximum hidden credit scaling of 16.0. The hidden credit scaling compensates for decreased hidden credit as described in Section 4.3. Even at a hidden credit scaling factor of 16.0, the Tree-Like Perceptron is still at a disadvantage since the number of outputs in the MLP is 26.

Effect of Hidden Credit Scaling

The effect of the hidden credit scaling on the training of both the architectures is dramatically displayed in Figures 4-4 and 4-5. In both architectures, use of the hidden credit scale factor resulted in an order of magnitude decrease in sum of squared error. In fact, the MLP converged to perfection in when run with an optimal hidden credit scaling factor of 8.0. If the scaling factor could be further increased, past the maximum of 16.0, the Tree-Like Perceptron could fully compensate for the 26 outputs which are present in the MLP.

Simulation Results and Discussion

In the 80 hours allocated, the MLP was able to run 32 training sessions of the unscaled algorithm.

The Tree-Like Perceptron was able to run 10 training sessions comprising 160 subnetworks. Figure 4-6 shows the number of subnetworks remaining to be retrained after each training session. Note that after the 80 hours only 6 out of the original 26 subnetworks did not train to perfection. Thus many of the network were able to converge in the allocated number of hidden units. Moreover, the subnetworks which had a low probability of convergence were identified quickly and the resources were focused on them. If an adaptive allocation scheme was also used, the remaining subnetworks may also have trained to perfection.

Figure 4-7 shows the error trajectories for the best simulations for each architecture. Note that for the Tree-Like Perceptron, the trajectory is composed of 26

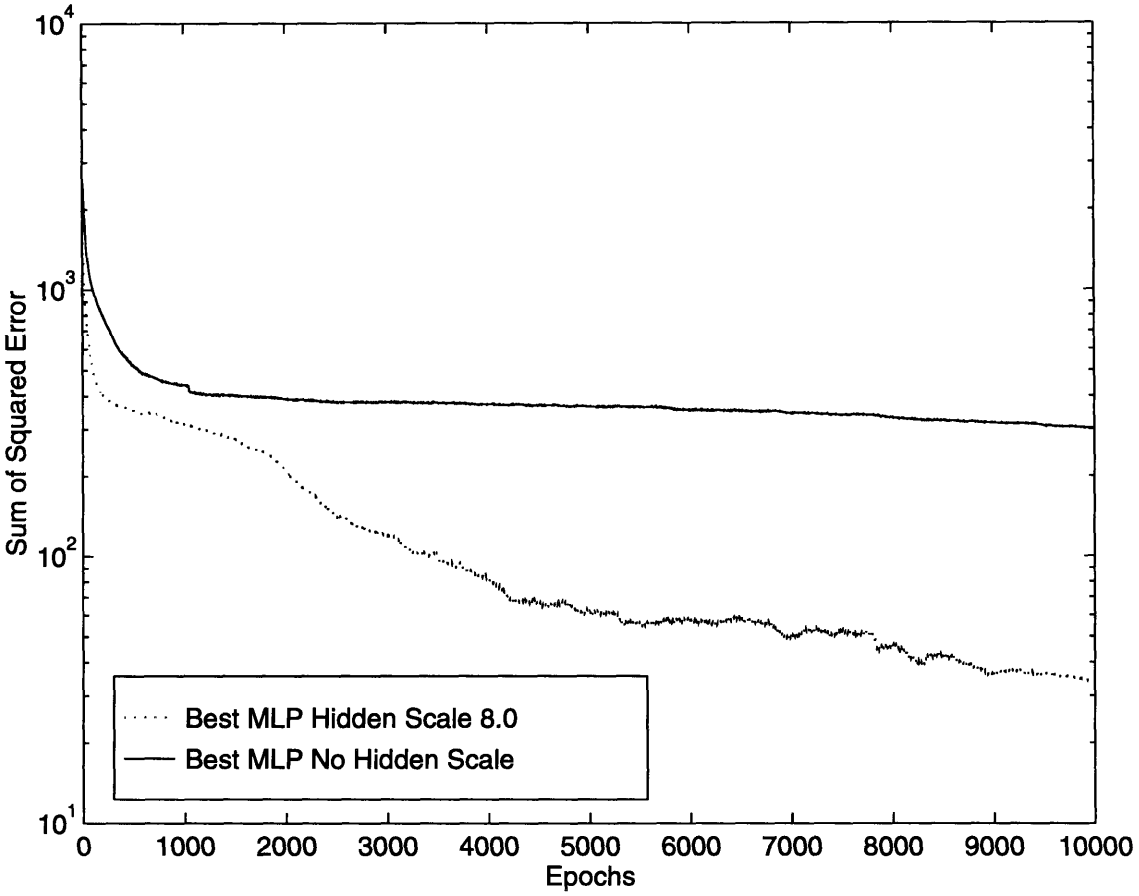


Figure 4-4: The MLP was run with and without hidden credit scaling. The MLP with scaling converged to zero classification error, whereas the best unscaled network had 42 % classification error. Sum of squared error trajectories are shown above. This simulation represents the Character Recognition task.

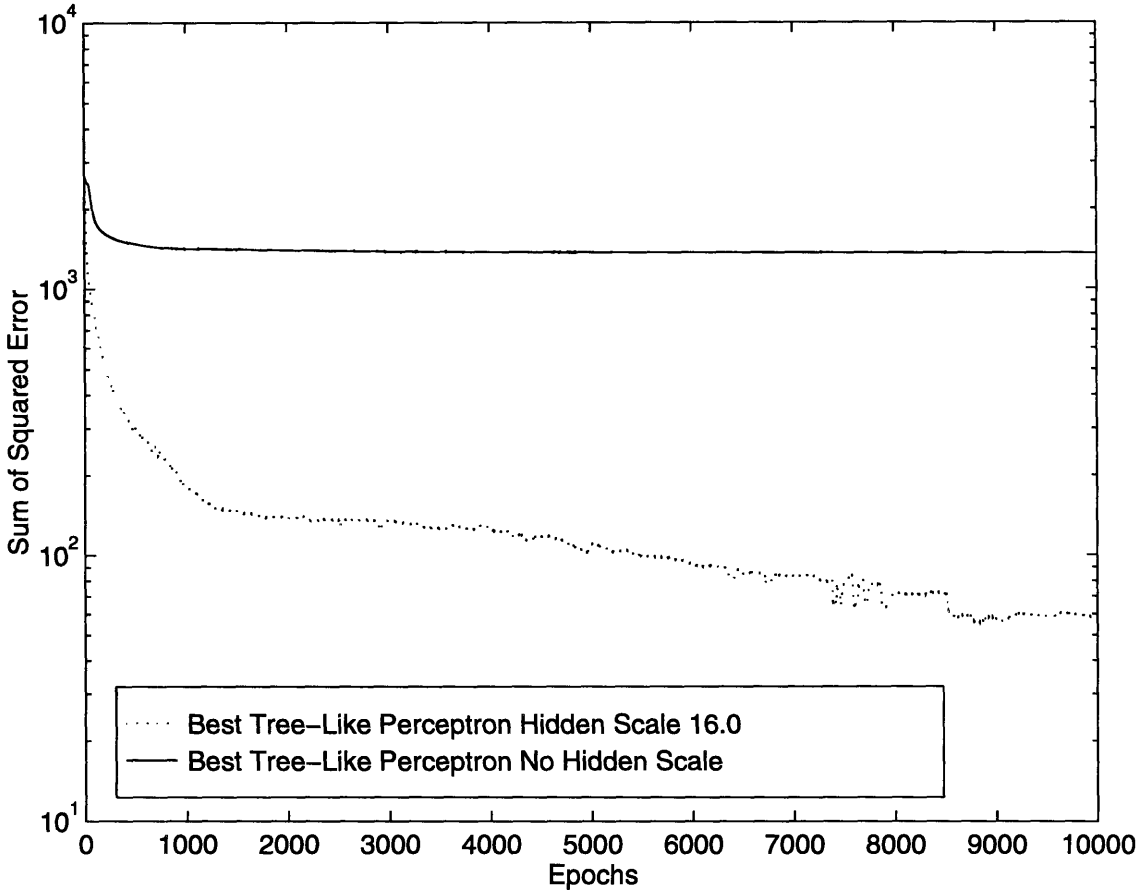


Figure 4-5: The Tree-Like Perceptron was run with and without hidden credit scaling. The Tree-Like Perceptron with scaling converged to $\approx 1\%$ classification error, whereas the best unscaled network had a worst case classification error of 100 %. Sum of squared error trajectories are shown above. This simulation represents the Character Recognition task.

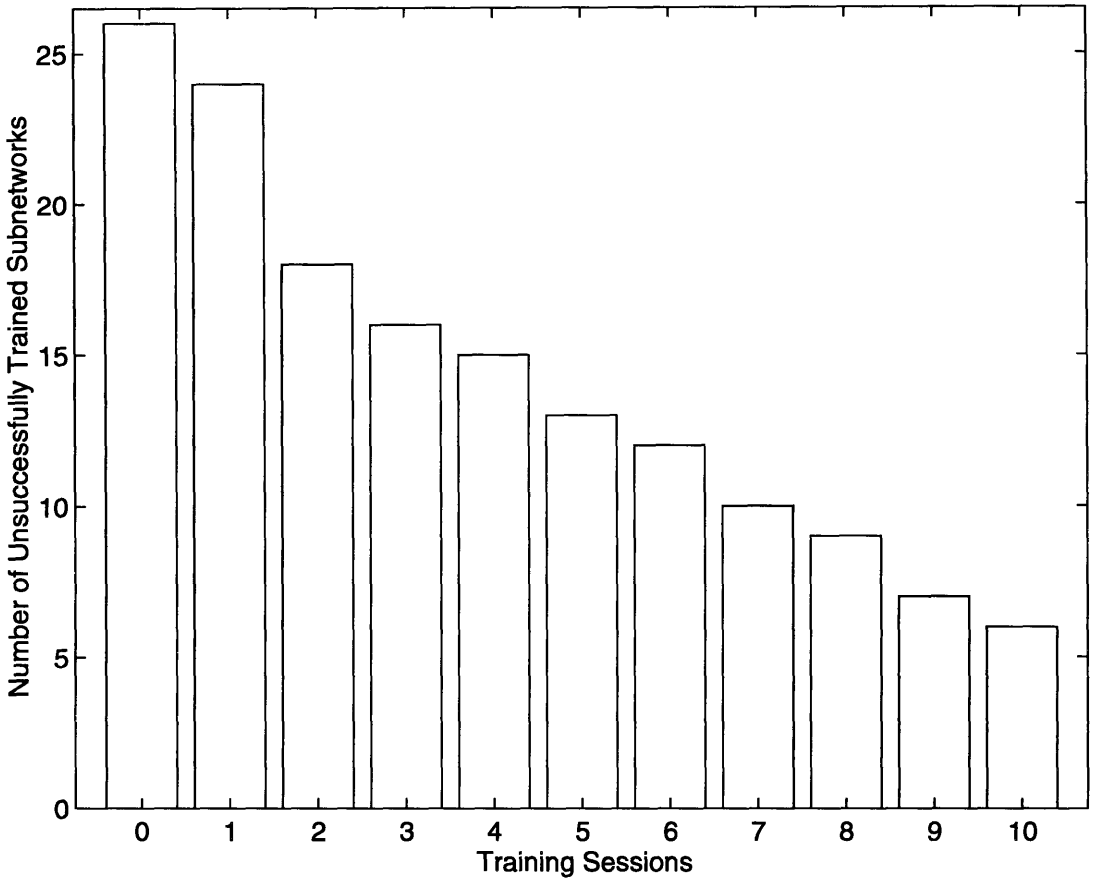


Figure 4-6: Remaining Tree-Like Perceptron subnetworks which have non-zero classification error after each training session. This simulation represents the Character Recognition task.

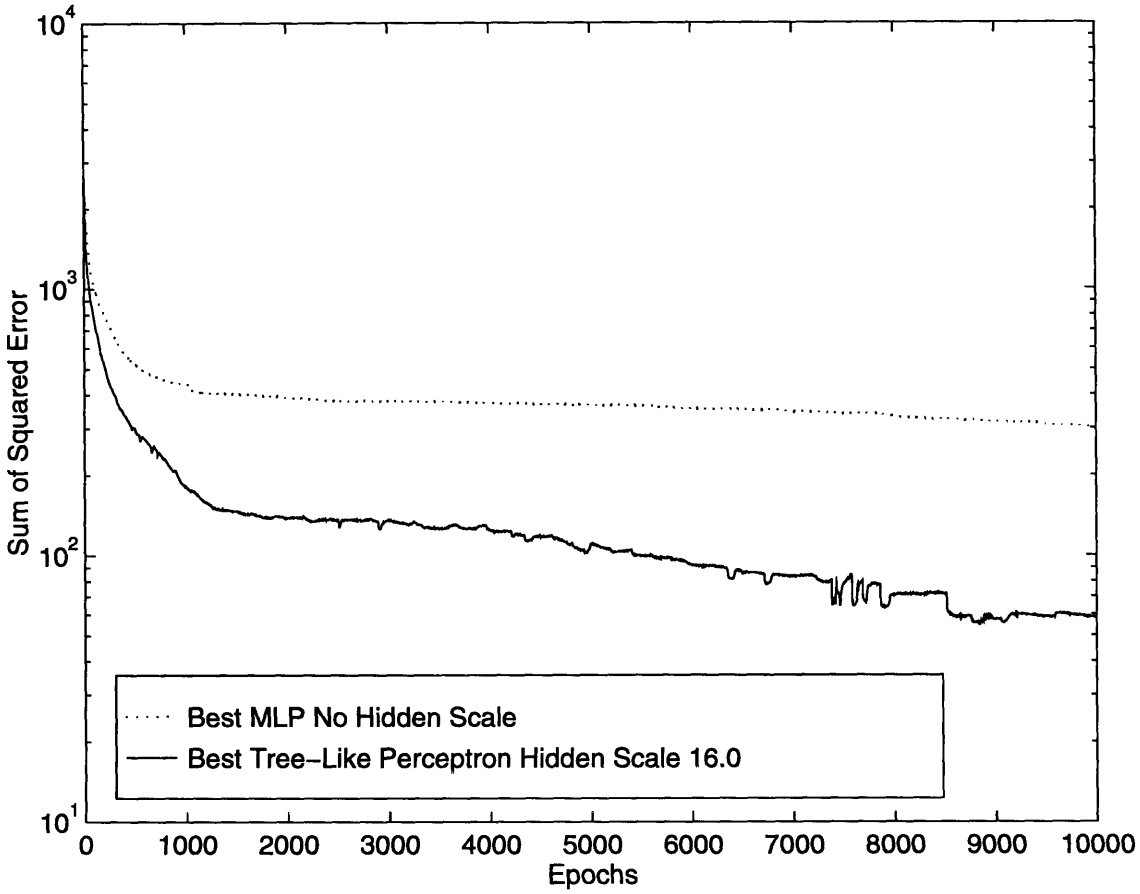


Figure 4-7: Error Trajectories for MLP versus Tree-Like Perceptron for the Character Recognition task. The *best* network was chosen according to the description in the text.

different subnetwork trajectories summed together. The most important aspect of this comparison is the difference in final sum of squared error. The Tree-Like Perceptron network clearly outperforms the unscaled MLP by almost one order of magnitude in final sum of squared error. If the comparison is done by classification error (Figure 4-8) a similar relationship can be seen. The best classification errors can be seen in Table 4.2. After the training sessions have completed, the best Tree-Like Perceptron network has $\approx 1\%$ classification error whereas the best unscaled MLP network has $\approx 42\%$ classification error !

The Tree-Like Perceptron network performed much better than the unscaled MLP. Using the decomposition strategy the classification error was reduced greatly as seen in Table 4.2. The Tree-Like Perceptron is disadvantaged in that the hidden units

Architecture	Best Classification Error
MLP unscaled	41.7 %
Tree-Like Perceptron scaled	1.2 %

Table 4.2: Best Classification Errors – Character Recognition. The classification errors for each network is given. The Tree-Like Perceptron classification is a worst case calculation.

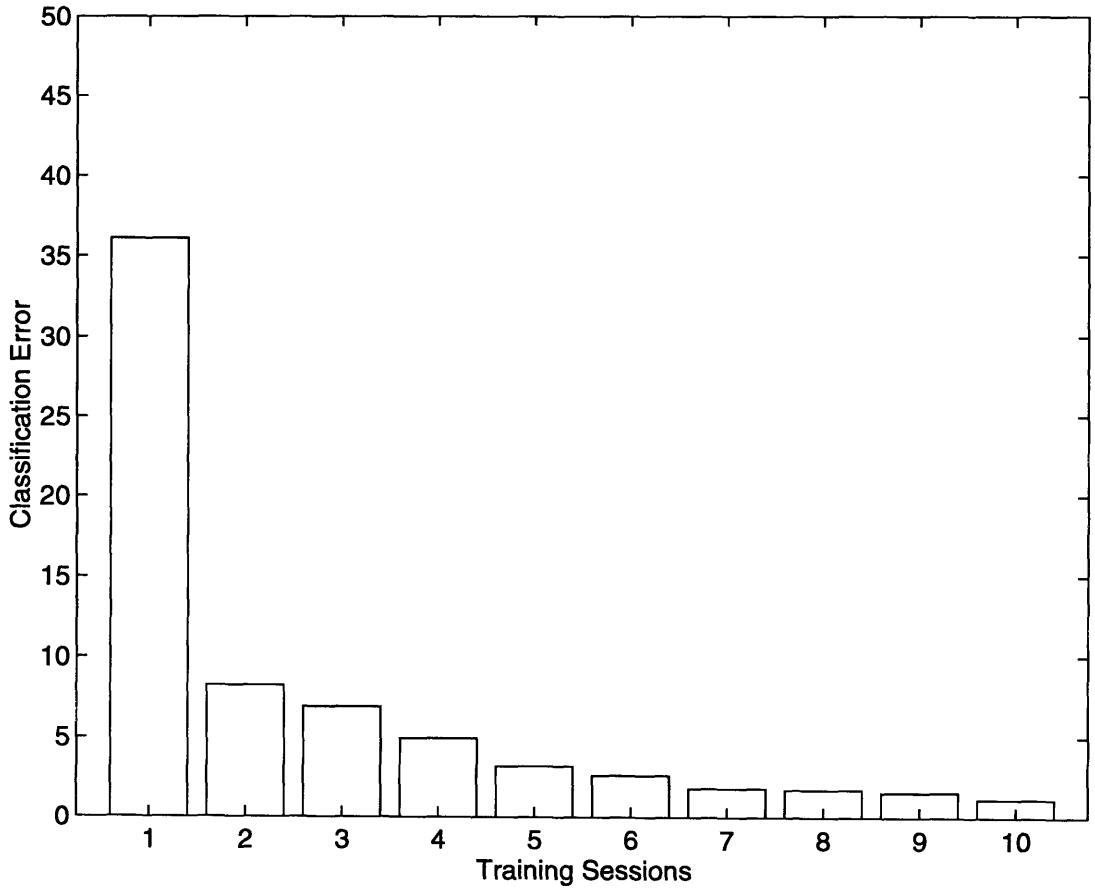


Figure 4-8: Classification Error for Tree-Like Perceptron network after each training session of the Character Recognition task. The scaled MLP achieved perfect classification. The best classification error achieved by the unscaled MLP was 41.7 %.

receive only a fraction of the error feedback the MLP enjoys. Thus a fair comparison should include a hidden credit scaling factor for the Tree-Like Perceptron to compensate for the difference (Section 4.3). This example has been *constructed* to make the performance conditions similar by using a hidden credit scaling factor of 16.0 for the TLP versus no hidden credit scaling for the MLP (which has an intrinsic scaling of 26 from the output units). Clearly, since the resources are available to achieve exact learning, scaled training of the MLP network would be used in practice. However, this simulation comparison shows that the hidden credit scaling if increased may increase the performance of the Tree-Like Perceptron also, as indicated by the increase in performance in Figure 4-5. Unfortunately, the limitations of the computing platform prevented such an increase in hidden credit scaling.

Both the simulation examples highlight the efficiency of the decomposition approach.

The Computer Speech example resulted in half the classification error when the decomposition approach was applied. Moreover, the decomposition approach allowed partial training of the network in the context of the resource limited environment. Unfortunately, the Computer Speech example was difficult to train and thus required a large number of parameters to achieve this level of performance.

The Character Recognition example represents a problem which can be solved by using the MLP with a hidden credit scaling factor. In practice, the exact solution would be used. However, in the context of an appropriate comparison, the decomposition strategy was efficient in reducing the network error. In this comparison, the Tree-Like Perceptron was able to reduce the classification error from $\approx 42\%$ to $\approx 1\%$.

The next chapter looks at improving the results of the Tree-Like Perceptron by increasing the intersubnetwork communication. The resulting architecture is a mix of the MLP feedforward computation and the Tree-Like Perceptron learning strategy.

Chapter 5

Tree Structured Neural Network Architectures - The Tree-Like Shared Network

5.1 Introduction

The previous chapter introduced the family of Tree-Like neural network architectures. The simplest member, the Tree-Like Perceptron, based on the Multilayer Perceptron (MLP), displayed good training performance when compared to the unscaled MLP in the Computer Speech and Character Recognition task. However, in both cases, the subnetworks were run in serial, since the computational resources did not allow for parallel training. Because of this limitation, few training sessions could be run. If the Tree-Like Perceptron could be adapted to the hardware dedicated for the MLP, then parallel training would be easier to achieve on existing platforms.

In this chapter, the discussion of the Tree Structured Architectures is concluded with a discussion of the Tree-Like Shared Network. The Tree-Like Shared network uses the concept of linear independence developed in Chapter 3 to create a network with parallel learning characteristics in an MLP-type architecture.

5.2 Tree-Like Shared Network

5.2.1 Architecture

The Tree-Like Shared network shares common structure with the Tree-Like Perceptron and the MLP. In feedforward computation, the Tree-Like Shared network is identical to the MLP. Subnetwork are still present as shown in Figure 5-1. For each output there is a subnetwork with a specified number of hidden units. The number of hidden units can be chosen as in the Tree-Like Perceptron. Unlike the Tree-Like Perceptron, the subnetworks are not independent, each subnetwork hidden layer is connected to every other subnetwork output by hidden to output weights (Figure 5-1).

The addition of hidden to output connections allows the output of each subnetwork to *use* the linearly independent vectors created in the internal representations of the other subnetworks (Chapter 3). Presumably, the number of required vectors in the internal representation of each subnetwork would decrease and thus converge faster.

In addition to the advantage gained by the *sharing* of linearly independent vectors, the architecture has still maintained a degree of parallelization in the learning process. The computation is identical to the MLP and thus the advantages of redundancy (Chapter 4) during utilization in an application is lost.

5.2.2 Training

The training of the Tree-Shared network is similar to the Tree-Like Perceptron in that each subnetwork is trained individually and in parallel. However, the additional hidden to output weights must also be updated. These updates are done with respect to the hidden unit to which the weight is connected.

A potential pitfall of this method is the early response of the subnetworks. Initially, the hidden unit outputs will be changing rapidly to adjust to the targets, as described in Section 3.4. As a result, the subnetworks will be responding to both the hidden units which they control and external units which will be providing changing values. These external hidden units can be thought of as disturbances to the subnet-

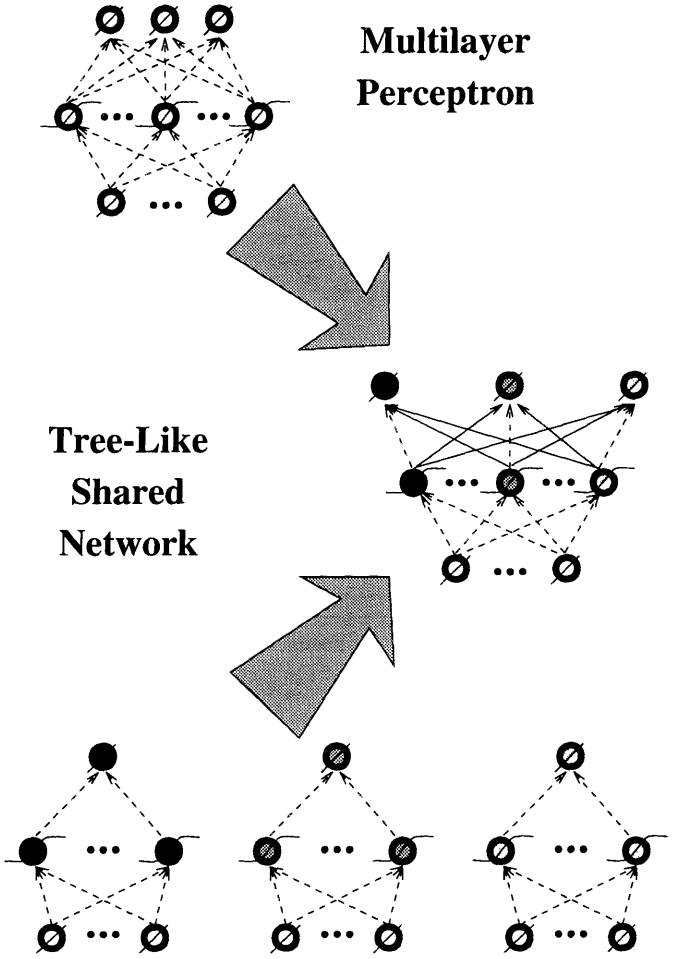


Figure 5-1: Dual origin of Tree-Like Shared network. The Tree-Like Shared network has computational aspects of both the MLP and Tree-Like Perceptron network. The feedforward computation is identical to the MLP, note the fully connected structure of the weights. The architecture, however, is still subdivided into subnetworks, designated by the shading of the output and hidden units. In this way, the parallel learning approach is maintained. The solid lines in the Tree-Like Shared network represent the hidden to output weights which allow sharing of the linearly independent vectors amongst the subnetworks. The solid lines are updated by the subnetwork *to which* it provides input.

work and may result in poor training early on. As the outputs of the hidden units stabilize, the subnetworks will adjust to the new hidden unit values. It is unclear, *a priori*, whether this will result in poor training performance.

5.3 Simulation Results and Discussion

The simulation example used for this comparison is the Character Recognition task used in the previous chapter. It is described in detail in Appendix C. The training set has 2600 input-output patterns. Each input has 64 attributes and each output has 26 attributes, all are binary valued.

The Tree-Like Shared network and MLP have similar computation times and as such, the comparison was normalized to number of training sessions. Each network was trained for 20 sessions. The MLP had 500 hidden units, the maximum for the Adaptive Solutions CNAPS Neurocomputer. The Tree-Like Shared network had 20 units per subnetwork, approximately 500 units distributed amongst the 26 subnetworks.

The training sessions were run at a learning rate of 0.01 with no momentum.

In analogy with the simulations of the Tree-Like Perceptron, the Tree-Like Shared was trained with a hidden credit scaling of 16.0 and the MLP was trained without a hidden credit scaling. For practical purposes, the MLP can achieve exact learning when a hidden credit scaling of 8.0 is used in training. However, the Tree-Like Shared is limited to a scaling factor of 16.0, which is smaller than the number of outputs in the network (Section 4.3). Thus the Tree-Like Shared is already at a disadvantage and by eliminating the hidden scaling factor for the MLP, an appropriate basis of comparison can be established.

The *best* training sessions were chosen on the basis of lowest final sum of squared error.

Figure 5-2 shows the training trajectories for the *best* MLP (both scaled and unscaled), Tree-Like Shared networks and Tree-Like Perceptron networks. The Tree-Like Perceptron results are from the simulation of previous chapter, and are included

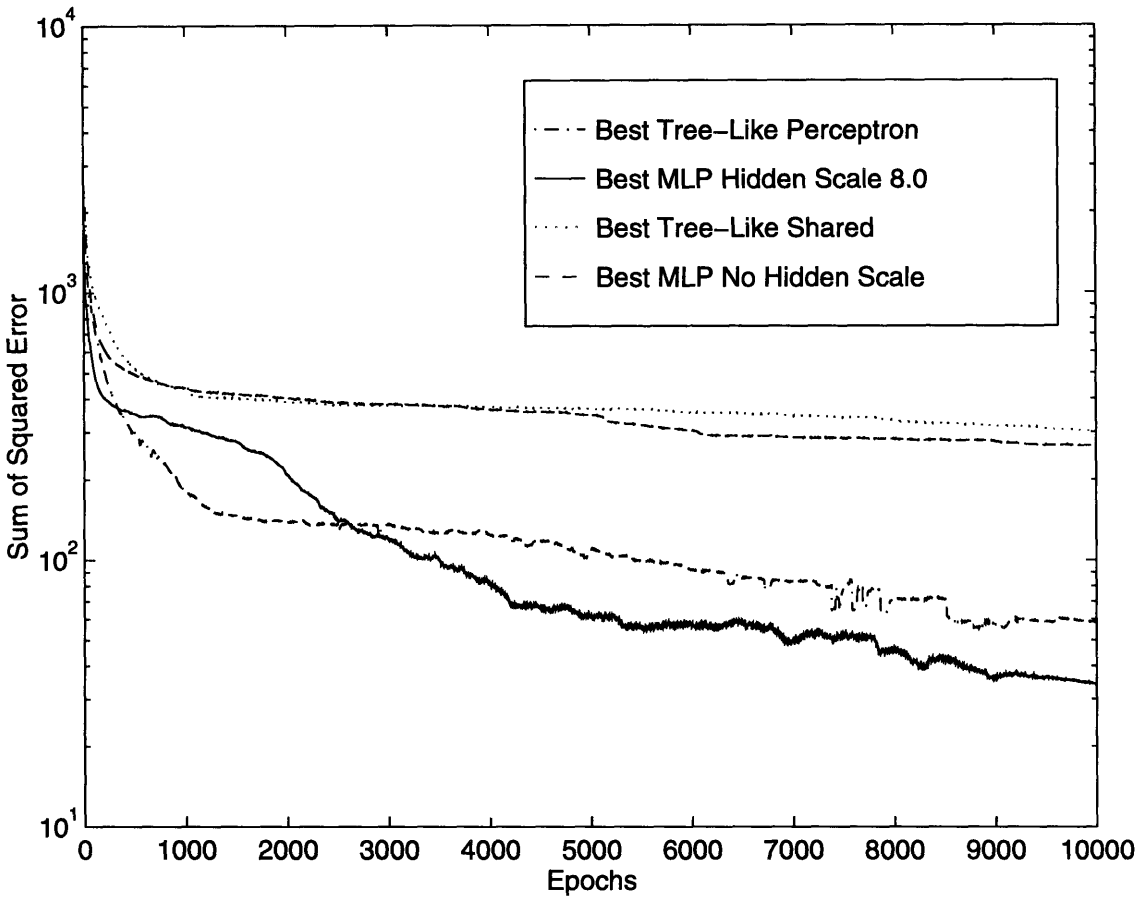


Figure 5-2: Error History for Tree-Like Shared Network versus Tree-Like Perceptron versus MLP. The *best* architecture was chosen on the basis of lowest final sum of squared error. The Tree-Like perceptron trajectory is taken from the simulations of the previous chapter.

to compare the two Tree-Like architectures. The scaled MLP results are also included to show the increase in learning created by the hidden credit scaling. Note how the unscaled MLP and Tree-Like Shared have similar training performance for sum of squared error.

The best classification errors are shown in Table 5.1. Even though the sum of squared error was similar, the classification error of the Tree-Like Shared network is approximately 25% lower when compared to the unscaled MLP.

In comparing the Tree-Like Shared network to the Tree-Like Perceptron, the Tree-Like Perceptron outperforms the Tree-Like Shared network in both sum of squared error (Figure 5-2) and classification performance.

Architecture	Best Classification Error
MLP (unscaled)	41.7 %
Tree-Like Shared	15.8 %

Table 5.1: Best Classification Errors – Character Recognition

The Tree-Like Shared network has shown to be a useful strategy in training. Although, the expected performance gain over the Tree-Like Perceptron was not achieved, the Tree-Like Shared network does have the advantage of parallel training on platforms designed specifically for the MLP architecture. That is, the Tree-Like Perceptron subnetwork training sessions were run in a serial fashion, whereas the Tree-Like Shared network was run with all subnetworks in parallel taking advantage of both the specialized hardware and parallel training.

The decrease in expected performance may be due to the changing internal representations early in the training process. Chapter 3 showed that the minimization of the sum of squared error modulated the change in internal representation towards exact learning. If the inputs from the other subnetworks are constantly changing, the effect on the subnetwork is that the target vector is changing (disturbance effect). As a result, the subnetwork has difficulty adjusting its internal representation to the changing target, resulting in decreased performance.

Showing the efficacy of a new neural network architecture or algorithm is an inductive process. It must be applied to example after example before it is clear that it is generally useful. Even with motivations such as parallel training and biologically relevance, it is unclear *a priori* how well an architecture will perform on an arbitrary application. The Tree Structured Architectures are an excellent example of this. The model indicates increased learning performance and the results are consistent with improved performance. However, further work with Tree-Like Shared network showed that the learning performance is not always predictable.

Unfortunately, neural network design principles, in the current form, represent a set of strategies which have few quantitative measures. As such, the use of intuition and raw computational power are likely to be the most effective design tools.

Chapter 6

A Novel Two Hidden Layer Architecture for Reduced Parameter Representations

6.1 Introduction

In evaluating neural network architectures, there are many important factors. In Chapters 4 and 5 the efficiency of computation and speed of training were the metrics of performance. Another important facet of neural network training is the size of the representation generated by the training process. Large representations use more storage space on the computing elements and are generally more computationally intensive to train [71].

The size of the representation is also related to the generalization ability of a network. Generalization refers to a network's ability to predict the correct output for input patterns that are not in the training set. In principle, a network can be trained on a subset of all the data, instead of the entire data set, and still produce the appropriate output for the remainder of the data set. This is important for applications where the all data is not known *a priori* or the data set is far too large to train a network in a reasonable amount of time. Generalization ability is related to the

number of parameters in a feedforward neural network (e.g. Multilayer Perceptron) through the Vapnik-Chervonenkis (VC) dimension [13, 32, 31, 8]. Simply put, the generalization ability of a network decreases as the number of weights in the network increases [15]. Thus the notion of a reduced parameter, or minimal, representation is appealing from the viewpoint of generalization.

The literature has many examples of algorithmic and analytical approaches to obtain minimal representations. Strategies to modify the number of parameters during training have been successful in achieving reduced parameter representations. Pruning algorithms start with some number of parameters and eliminate those which do not *contribute* to the output significantly [69, 32]. Construction algorithms start from a set of small of parameters, adding parameters as needed [19, 60]. Other training algorithms use both the elimination and addition of weights to produce a reduced parameter representation [3].

Bounds on the number of hidden units required to guarantee exact learning have also provided useful results in the context of reducing the number of parameters in a representation [34, 7, 4]. Section 3.3 describes the minimal number of hidden units required for learning an arbitrary target. The determination of the *minimal* number of hidden units required for a fixed set of patterns (as opposed to arbitrary) is an open question. The minimal number of hidden units would correspond to the optimal architecture for size and generalization for a feedforward neural network.

This investigation studies a novel two hidden layer architecture developed to reduce the number of parameters in MLP-like networks. The analytic foundations of this architecture have been developed from results in Chapter 3. The motivation and structure of the architecture are described and simulations versus Multilayer Perceptron show the relative learning performance.

6.2 Novel Two Hidden Layer Architecture

This section describes the analytic development of the novel two hidden layer architecture.

Lemma 3.2 shows that there exists a linear combination of a set of input vectors, from distinct input patterns, which results in a vector of distinct elements. The input vectors are the input patterns of a training set put into a node-wise orientation (Section 3.2). Alternatively, Lemma 3.2 states there exists weights such that distinct input patterns can be fed into a single linear node (Figure 6-1 Network A) and the output of that linear node is different for each input pattern.

This single element *coding* and the target patterns can now be combined to form a new single input training set. This new training set can be trained on a single hidden layer MLP with one input (Figure 6-1 Network B).

The combination of these subnetworks (Figure 6-1 Networks A and B) gives a novel two hidden layer network (Figure 6-1 Network C). This new network still has the universal approximation characteristics of a conventional MLP as a result of Lemma 3.2. Also, the network has the same bounds on hidden units as a conventional MLP, except that these bounds apply to the second hidden layer. But most importantly, this network has fewer parameters than a conventional MLP for many application-relevant network sizes.

A conventional MLP with N_I input units, N_H hidden units and N_o output units has :

$$(N_I + 1) \cdot N_H + (N_H + 1) \cdot N_o \Rightarrow (N_I + N_o + 1) \cdot N_H + N_o$$

parameters, whereas the new architecture with the same parameters would have :

$$N_I + 1 + 2 \cdot N_H + (N_H + 1) \cdot N_o \Rightarrow (N_o + 2) \cdot N_H + N_I + N_o$$

parameters. The difference in number of parameters (MLP - novel) is $(N_I - 1) \cdot N_H - N_I$. This implies that the MLP has fewer parameters when :

$$N_H < \frac{N_I}{N_I - 1}$$

which for $N_I > 2$ is $N_H = 1$. Thus the MLP has fewer parameters only when $N_H = 1$, meaning that for most networks, the novel two hidden layer architecture has fewer

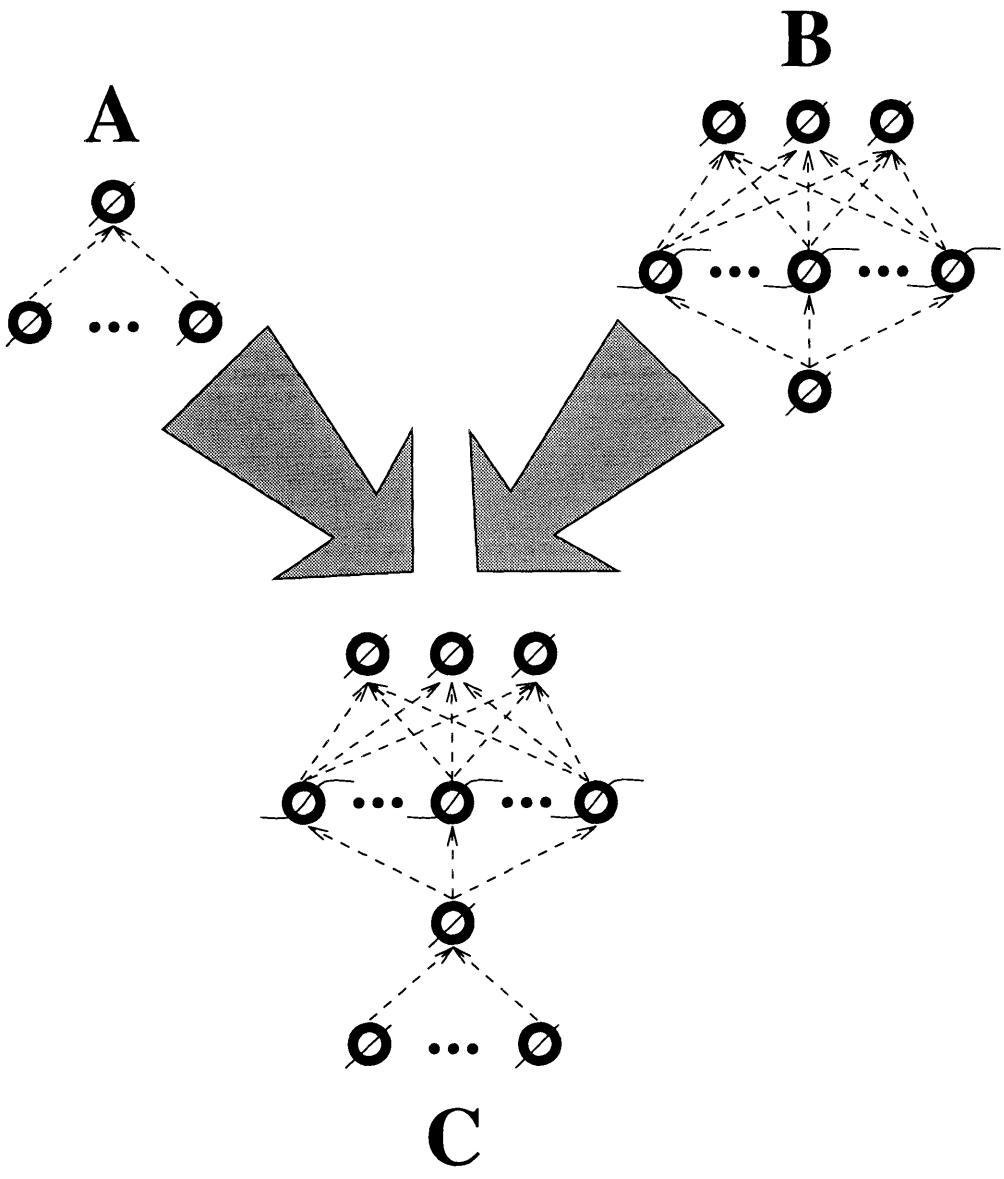


Figure 6-1: The Novel Two Hidden Layer Network. The two hidden layer network, **C**, is comprised of a coding portion, **A**, which produces a distinct element for each input pattern and a single input single hidden layer MLP portion, **B**, to perform the mapping to the target.

parameters.

Theoretically, the new architecture can represent any training set that the MLP can, however, training the new architecture may be very difficult. Also, many computers are limited in precision and dynamic range and thus a linear unit is unsuitable. Since the hyperbolic tangent function is bijective (i.e. 1:1) it would preserve the uniqueness in the single element encoding. Due to these practical concerns the simulations below were run with multiple sigmoidal *coding* units in the first hidden layer. The number of coding units in the first hidden layer was chosen such that the total number of parameters was still fewer than in the MLP.

6.3 Simulation Results

The Character Recognition task used in this simulation is described in detail in Appendix C. For the purposes of this discussion, the example contains 2600 input-output examples. Each input has 64 attributes and each output has 26 attributes. All the attributes are binary valued. Both networks were trained on an Adaptive Solutions CNAPS Neurocomputer using the backpropagation algorithms (one layer and two layer) that come as part of the machine software. For a discussion of the Adaptive Solutions CNAPS Neurocomputer refer to Appendix D. The MLP was run with 100 hidden units. The novel architecture was run with 100 units in its second hidden layer and 10, 20 or 40 coding units in its first hidden layer. The learning rate used for both architectures was 0.01 and no momentum was used.

Five training sessions for each network were run and the session ending in the lowest sum of squared error was used for comparison.

Simulation results in Figure 6-2 show poor performance of the novel architecture for all coding layer sizes. The classification error for all the novel architecture networks was 100 % error, meaning the architecture did not learn any patterns.

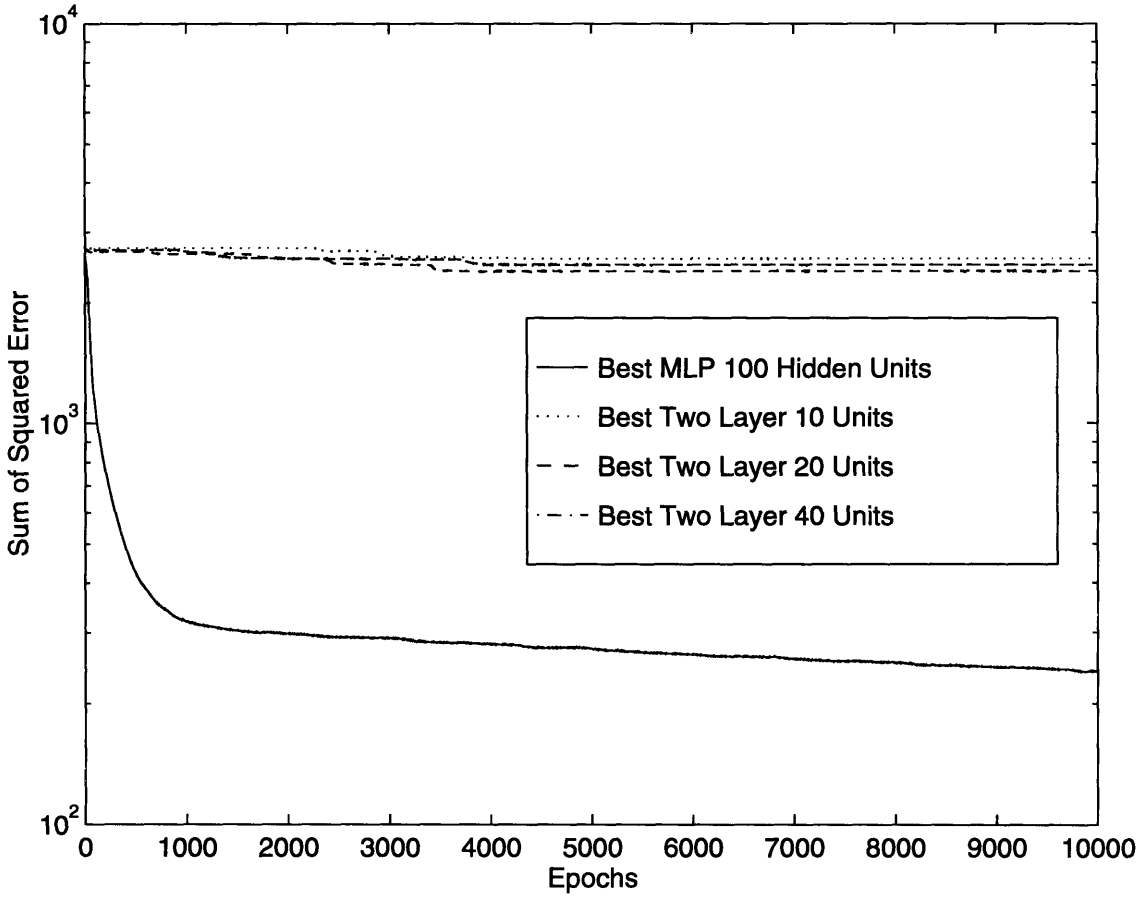


Figure 6-2: Error Trajectories for Novel Two Hidden Layer Network versus MLP. For each network, the error trajectory with the lowest sum of squared error for each architecture was chosen from five training sessions.

6.4 Discussion

The novel architecture discussed here has been motivated by a reduction in the number of parameters in feedforward neural network representations. Using results developed in Section 3.3, a novel two hidden layer is described with equivalent representation ability, in principle, as the MLP. However, the training algorithm used was unable to train the weights to any substantial decrease in sum of squared error (Figure 6-2). Poor performance is likely due to a high degree of sensitivity of the model to precision and dynamic range. For example, the exchange of the sigmoidal units for the linear coding units severely limited the dynamic range of the coding layer. In fact, the distinct coding may have been impossible to generate. The limited precision and dynamic range of the CNAPS Neurocomputer exacerbated this difficulty. Further training in environments with higher precision and dynamic range may show this network to be a useful reduced parameter representation.

Chapter 7

Conclusions

In an attempt to design new algorithms and architectures for faster training and computational efficiency of feedforward artificial neural networks, the analysis of learning dynamics has been important.

The concept of a system matrix and its condition number led to the development of the linearly independent internal representation. The linearly independent internal representation was useful in determining the minimum architecture of an MLP required for exact learning. In addition, the minimization of the sum of squared error was shown to enhance the linear independence of the internal representation, thereby increasing the likelihood of achieving exact learning. As a result, the linearly independent internal representation was a promising substrate for algorithm and architecture design.

The Tree-Like Architectures used a decomposition strategy, shown to be both computationally efficient and biologically relevant, to increase training performance in the MLP. The Tree-Like Perceptron was able to show the utility of partial training in the context of a resource limited environment, in training the Computer Speech example. The classification error was halved at the expense of a greater number of network parameters. The Tree-Like Perceptron when compared to the MLP under similar training conditions, on a Character Recognition task, was able to decrease the classification error from $\approx$ 42% to $\approx$ 1%. Thus showing the efficacy of the decomposition approach. Unfortunately, the use of a hidden scale factor in training

the MLP, allowed the MLP to train to perfection. Limitations on the size of the scale factor available made it impossible to carry an appropriate comparison to the MLP which trained to perfection.

Preliminary results from the Computer Speech and Character Recognition examples indicate the Tree-Like Perceptron to be potentially useful in the efficient training of large-scale neural networks.

The application of the linear independence concept to the Tree-Like Perceptron produced the Tree-Like Shared network, which outperformed the MLP in classification error on the Character Recognition example. However, the Tree-Like Shared network was expected to outperform the Tree-Like Perceptron due to the increased sharing of internal representation vectors. Since the Tree-Like Perceptron actually outperformed the Tree-Shared network, it is likely that the disturbance effect of the changing internal representations in other subnetworks, actually slows down convergence. The Tree-Like Shared network has the advantage of running a parallel training scheme within the confines of specialized hardware for the MLP.

Some potential pitfalls of the Tree-Like Architectures may result from the underestimated efficacy of the MLP and backpropagation. The simple addition of the hidden credit scaling factor was able to boost the Character Recognition MLP network to perfection. It is likely that for certain applications ad hoc modifications to the backpropagation algorithm are likely to outperform the Tree-Like Architecture strategy. Delineating the efficacy of the Tree-Like Architecture is an important measure for practical utilization of the decomposition strategy. A measure of this may be the critical ratio, which can indicate the utility of the Tree-Like Architecture over the MLP. However, the critical ratio is, currently, difficult to calculate.

A novel two hidden layer architecture derived directly from the concept of linear independence was designed specifically to reduce the number of parameters in the learned representation. A reduced number of parameters decreases the storage requirement as well as increasing the generalization ability. Unfortunately, the two hidden layer architecture performed poorly. The poor performance may have been a result of the limited precision and dynamic range of the computing platform rather

than any inherent restriction in the network representation ability. Further work in more computationally robust environments may reveal the practical training characteristics of this reduced parameter network.

This investigation has identified some important principles in the design of new neural network architectures and algorithms. The linearly independent internal representation was shown to be necessary for exact learning and an inherent property of many current training strategies. The decomposition strategy was shown to be effective in reducing training and classification error in large-scale networks as well as being biologically relevant and computationally efficient. In addition, the reduction in size of parameterizations may be an important step in reducing storage requirements in architectures as well as increasing generalization ability.

As yet, it is unclear if simple ad hoc modifications to current strategies will solve the increased computational burden created by growing neural network application size. If these ad hoc modifications fail, as they are likely to do, the principles developed and demonstrated in this investigation will be important in the design of new computationally efficient and biologically relevant artificial feedforward neural network architectures and algorithms.

Chapter 8

Future Work

Many concepts presented in this investigation will be useful to future investigations in the design of new algorithms and architectures for learning and computational efficiency.

The concept of the linearly independent internal representation has been important in identifying the upper bound on the number of hidden units required for exact training. Further work may reveal an algorithm for determining the *minimal* number of hidden units required for exact learning of a fixed (rather than arbitrary) training set.

The decomposition strategy of the Tree-Like Architectures was tested on two examples. Although, the results showed potential benefits, the development of the critical ratio may aid in identifying problems for which the decomposition strategy will result in efficient learning.

Parameter reduction is also an important aspect of computational efficiency. The further development of the linear independence concept may aid in understanding the feasibility of reduced parameter representations.

Appendix A

Proofs

A.1 Proof of Lemma 3.1

Proof:

The proof consists of three cases: one where the elements of $\mathbf{s}$, denoted $[\mathbf{s}]_i, i = 1, \ldots, N$, have the same sign as the corresponding elements of $\mathbf{u}$, denoted $[\mathbf{u}]_i, i = 1, \ldots, N$; another where the elements are of opposite signs; and the last case where some of the signs may differ. Without loss of generality assume that $\|s\| = 1$.

Case 1 : $\text{sgn}([\mathbf{s}]_i) = \text{sgn}([\mathbf{u}]_i), i = 1, \ldots, N$

Noting that tanh is symmetric, choose $a > 0$ and $b = 0$ so that $[\mathbf{s}]_i \cdot [\mathbf{u}]_i \geq 0$. Since for all i $\exists\, [\mathbf{s}]_i \neq 0$ the resulting inner product is strictly positive, i.e. greater than zero.

Case 2 : $\text{sgn}([\mathbf{s}]_i) = -\text{sgn}([\mathbf{u}]_i), i = 1, \ldots, N$

Using a similar argument as in Case 1, choose $a < 0$ (since the signs differ) and $b = 0$ to obtain a strictly positive inner product.

Case 3 : Some but not all signs of $[\mathbf{s}]_i$ and $[\mathbf{u}]_i$ differ

This case generates three new subcases based on the following subdivision. Subdivide the set of indices $\{1, \ldots, N\}$ into two subsets. The first, I_{same}, denotes the indices for which $[\mathbf{s}]_i$ have the same sign as the

corresponding $[\mathbf{u}]_i$. The second, I_{diff}, are the remaining indices, for which the signs differ.

Subcase i :

$$\sum_{i \in I_{same}} |[\mathbf{s}]_i| - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| > 0$$

Choose $\epsilon > 0$ such that

$$\epsilon = \frac{1}{N_{same} + 1} \left(\sum_{i \in I_{same}} |[\mathbf{s}]_i| - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| \right)$$

where N_{same} is the number of elements in I_{same}.

Since $|\tanh(a[\mathbf{u}]_i)| < 1$, choose $a > 0$ such that $|\tanh(a[\mathbf{u}]_i)| > 1 - \epsilon, i \in I_{same}$ and $b = 0$.

The following inequalities result:

$$\begin{aligned} \sum_{i \in I_{same}} [\mathbf{s}]_i \tanh(a[\mathbf{u}]_i) &\geq \sum_{i \in I_{same}} |[\mathbf{s}]_i|(1 - \epsilon) \\ \sum_{i \in I_{diff}} [\mathbf{s}]_i \tanh(a[\mathbf{u}]_i) &\geq -\sum_{i \in I_{diff}} |[\mathbf{s}]_i| \end{aligned}$$

.

Combining the sums gives,

$$\begin{aligned} \mathbf{s}^T \mathbf{tanh}(a\mathbf{u}) &= \sum_{i=1}^{N} [\mathbf{s}]_i \tanh(a[\mathbf{u}]_i) \\ &= \sum_{i \in I_{same}} [\mathbf{s}]_i \tanh(a[\mathbf{u}]_i) + \sum_{i \in I_{diff}} [\mathbf{s}]_i \tanh(a[\mathbf{u}]_i) \\ &\geq \sum_{i \in I_{same}} |[\mathbf{s}]_i|(1 - \epsilon) - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| \\ &\geq \sum_{i \in I_{same}} |[\mathbf{s}]_i| - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| - \sum_{i \in I_{same}} |[\mathbf{s}]_i|\epsilon \\ &\geq (N_{same} + 1)\epsilon - N_{same}\epsilon \\ &\geq \epsilon \end{aligned}$$

Subcase ii :

$$\sum_{i \in I_{same}} |[\mathbf{s}]_i| - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| < 0$$

This case is similar to Subcase i, except that the roles of I_{diff} and I_{same} are interchanged.

Subcase iii :

$$\sum_{i \in I_{same}} |[\mathbf{s}]_i| - \sum_{i \in I_{diff}} |[\mathbf{s}]_i| = 0$$

Let $\epsilon > 0$, choose $\bar{b}$ such that

$$\begin{aligned} |\bar{b}| &= 2 \cdot \min_{k=1,\ldots,N} |[\mathbf{u}]_k| + \epsilon \\ \bar{k} &= \arg \min_{k=1,\ldots,N} |[\mathbf{u}]_k| \end{aligned}$$

Let the sign of $\bar{b}$ be opposite to the sign of $[\mathbf{u}]_{\bar{k}}$

Define $\bar{\mathbf{u}} = \mathbf{u} + \bar{b}\mathbf{1}$. For $\bar{\mathbf{u}}$ define $\bar{I}_{same}$ and $\bar{I}_{diff}$ as before. Thus,

$$\sum_{i \in \bar{I}_{same}} |[\mathbf{s}]_i| - \sum_{i \in \bar{I}_{diff}} |[\mathbf{s}]_i| \neq 0$$

and can use subcase i or ii to solve for the appropriate value of a. The required value of b is then $a\bar{b}$.

□

A.2 Proof of Lemma 3.2

To begin a definition is required,

Definition A.1 *A set of column vectors,* $\{\mathbf{s}_1, \ldots, \mathbf{s}_k\}$ *is said to have* distinct rows *if the matrix*

$$\mathcal{S} = [\mathbf{s}_1 | \ldots | \mathbf{s}_k]$$

has no identical rows.

When applied to input vectors, Definition A.1 is equivalent to the distinct input pattern condition. Lemma A.1, which follows, is used to extend Theorem 3.1 to

multiple inputs.

Lemma A.1 *Given a set of vectors,* $\{\mathbf{s}_1, \ldots, \mathbf{s}_k\}$, *with* distinct **rows**, $\exists$ $\{a_1, \ldots, a_k\}$ *such that*

$$\mathbf{u} = \sum_{i=1}^{k} a_i \mathbf{s}_i$$

has distinct **elements**.

Note that Lemma A.1 is simply a restatement of Lemma 3.2 without connotation to MLP networks.

Proof:

For $k = 3$, have $\mathbf{s}_1$, $\mathbf{s}_2$ and $\mathbf{s}_3$ with distinct rows. a_1 and a_2 will be found such that $a_1\mathbf{s}_1 + a_2\mathbf{s}_2$ has distinct elements and has distinct rows with respect to $\mathbf{s}_3$.

Let $\mathbf{q} = \mathbf{s}_1 + \mathbf{s}_2$ and

$$\delta = \min_{\substack{(i,j) \\ i \neq j}} |[\mathbf{q}]_i - [\mathbf{q}]_j|$$

That is, δ is the smallest difference between the elements of $\mathbf{q}$.

Let $\mathcal{A}$ be the set of ordered pairs (i, j) such that $|[\mathbf{q}]_i - [\mathbf{q}]_j| = \delta$.

There exist three possible cases,

Case 1 : $\delta \neq 0$ and $\mathbf{q}$ and $\mathbf{s}_3$ have distinct rows

That is, $\mathbf{q}$ has distinct elements, thus $a_1 = 1$ and $a_2 = 1$.

Case 2 : $\delta \neq 0$ and $\mathbf{q}$ and $\mathbf{s}_3$ do not have distinct rows

Let $\mathcal{B}$ be the set of indices i such that $[\mathbf{q}]_i = [\mathbf{s}_3]_i$.

Let

$$\hat{\gamma} = \min_{i \notin \mathcal{B}} |[\mathbf{s}_3]_i - [\mathbf{q}]_i|$$

and

$$\hat{\delta} = \min_{\substack{(i,j) \\ i \neq j \\ (i,j) \notin \mathcal{A}}} |[\mathbf{q}]_i - [\mathbf{q}]_j|.$$

That is, $\hat{\gamma}$ is the next largest difference between corresponding elements of $\mathbf{s}_3$ and $\mathbf{q}$. $\hat{\delta}$ is the next largest difference between the the elements of $\mathbf{q}$. Let

$$\hat{\rho} = \max_{(i,j)\in\mathcal{A}} \max([\mathbf{s}_2]_i, [\mathbf{s}_2]_j),$$

that is $\hat{\rho}$ is the largest element of $\mathbf{s}_2$, and let

$$\hat{\epsilon} = \frac{\min(\hat{\delta}, \hat{\gamma})}{\hat{\rho}+1}.$$

then $\mathbf{s}_1 + (1-\hat{\epsilon})\mathbf{s}_2$ has distinct elements and has distinct rows with respect to $\mathbf{s}_3$.

Case 3 : $\delta = 0$

Let

$$\hat{\delta} = \min_{\substack{(i,j) \\ i\neq j \\ (i,j)\notin\mathcal{A}}} |[\mathbf{q}]_i - [[\mathbf{q}]_j|.$$

That is, $\hat{\delta}$ is the next largest difference between the the elements of $\mathbf{q}$. Let

$$\rho = \max_{(i,j)\in\mathcal{A}} \max([\mathbf{s}_2]_i, [\mathbf{s}_2]_j),$$

that is ρ is the largest element of $\mathbf{s}_2$, and let

$$\epsilon = \frac{\hat{\delta}}{\rho+1}.$$

then $\mathbf{s}_1 + (1-\epsilon)\mathbf{s}_2$ has distinct elements. If $\mathbf{s}_1 + (1-\epsilon)\mathbf{s}_2$ does not have distinct rows with respect to $\mathbf{s}_3$ then follow case 2 with $\mathbf{s}_2 = (1-\epsilon)\mathbf{s}_2$.

Assume the proposition true for $k = r$, then truth for $k = r+1$ must be shown.

By assumption, $\exists\ \{\hat{a_1}, \ldots, \hat{a_r}\}$ such that $\hat{\mathbf{u}} = \sum_{i=1}^{r} \hat{a_i}\mathbf{s}_i$ has distinct elements and the vectors $\hat{\mathbf{u}}$ and $\mathbf{s}_{r+1}$ have distinct rows.

This is the $k = 3$ case without the distinct rows requirement again and b_1 and b_2 can be found such that $b_1\hat{\mathbf{u}} + b_2\mathbf{s}_{r+1}$ has distinct elements.

Letting

$$\begin{aligned} a_i &= b_1 \hat{a}_i,\ i = 1, \ldots, r \\ a_{r+1} &= b_2 \end{aligned}$$

$\sum_{i=1}^{r+1} a_i \mathbf{s}_i$ has distinct elements.

The proof follows by induction.

□

A.3 Proof of Proposition 3.1

Lemma A.2 *Given a set of linearly independent vectors,* $\{\mathbf{v}_1, \ldots, \mathbf{v}_k\}$ *with distinct patterns where* $\mathbf{v}_i \in \Re^N, i = 1, \ldots, k, \exists\ \{c_1, \ldots, c_k\}$ *and* $\{(a_1, b_1), \ldots (a_{N-k}, b_{N-k})\}$ *such that*

$$\mathbf{u} = \sum_{i=1}^{k} c_i \mathbf{v}_i$$

has distinct elements and

$$\{\mathbf{v}_1, \ldots, \mathbf{v}_k, \mathbf{tanh}(a_1\mathbf{u} + b_1\mathbf{1}), \ldots, \mathbf{tanh}(a_{N-k}\mathbf{u} + b_{N-k}\mathbf{1})\}$$

forms a complete basis of $\Re^N$.

Proof:

A vector of distinct elements, $\mathbf{u}$, can be generated by Lemma 3.2. Corollary 3.1 proves complete basis for $\Re^N$ can be generated with N non-linear units and a vector of distinct elements. If k linearly independent vectors are given, then choose $N - k$ vectors from the complete basis to create another complete basis which includes the k given linearly independent vectors. This follows from the fact that the number of linearly independent vectors required to form a complete basis is the dimension of the vector space.

□

Proof of **Proposition 3.1:**

Lemma A.2 shows that given the linearly independent vectors in the input data, number of nonlinear hidden units required to form a complete basis can be reduced from N_P. Let $k = \mathcal{R}([\mathbf{v}_1|\ldots|\mathbf{v}_{N_I}|\mathbf{1}])$. Choose k linearly independent vectors from the input data (and $\mathbf{1}$). Assume a network with N_P hidden units as described in Theorem 3.2 already exists. By Lemma A.2, choose $N_P - k$ vectors from the complete basis of the existing network. If the k hidden units not chosen for the basis are eliminated and the k linearly independent input vectors are connected directly to the output, then the result is a network with $N_P - k$ hidden units with a complete basis consisting of vectors of the internal representation and input vectors. Moreover, any additional hidden units would be unnecessary since a complete basis already exists, the deletion of any node would result in an incomplete basis. Thus $N_P - k$ hidden units are enough to interpolate an arbitrary data set.

□

A.4 Proof of Corollary 3.2

Proof:

By adding multiple output nodes, the number of target vectors is increased. Each target vector is an element of $\Re^{N_P}$. Thus the complete basis due to the $N_P - \mathcal{R}\left(\left[\begin{array}{c|c|c|c} \mathbf{v}_1 & \ldots & \mathbf{v}_{N_I} & \mathbf{1} \end{array}\right]\right)$ hidden units and the linearly independent input vectors can span all the required target vectors. Note that each new output node adds new hidden to output and input to output weights. Each set of weights can be set independently of other output nodes to form the appropriate linear combination for the particu-

lar target vector required. As a result, any number of output nodes can be supported, since the complete basis spans $\Re^{N_P}$.

□

Appendix B

Observations on "Characterization of Training Errors in Supervised Learning Using Gradient-Based Rules"

The following is an excerpt from the communication [66] outlining an erroneous proof in Wang and Malakooti [74]. As detailed below a similar version of Proposition 3.1 (Section 3.3) as stated in the above article follows from an erroneous proof.

B.1 Letter to the Editor of *Neural Networks*

Editor:

The recently published article *Characterization of Training Error in Supervised Learning Using Gradient-Based Rules* by Jun Wang and B. Malakooti [74], represents a significant effort in describing training errors in feedforward neural networks. This paper uses a systems-theoretic framework to outline some appealing results. Unfortunately, as we will describe, some of the proofs are incomplete, indicating that the results may be less tractable than initially conjectured.

In particular, Theorem 3 (which provides a basis for many statements that follow)

relates the rank of the Jacobian matrix of the neural network to the steady-state error during training. Specifically it states

> "... a steady-state training error $E[\bar{w}]$ is always an absolute minimum error iff the rank of the matrix $\Psi(x, \bar{w})$ is Pm."

where $\Psi(x, \bar{w})$ is a matrix composed of the Jacobian matrix of each output node transposed. It is important to note that the gradient direction, $-\eta(t)\nabla_w E$, is given by $\eta(t)\Psi(x, w)\tilde{e}(y)$, where $\eta(t)$ is a time-dependent learning rate, $\tilde{e}(y)$ is the vector of errors for each pattern and y the network output. An absolute minimum is defined by the authors as a zero error minimum.

The proof as published verifies the sufficiency condition, but fails to address all the aspects of the necessary condition.

The authors' sufficiency proof follows from a direct linear algebraic argument. That is, if $\Psi(x, w)$ has full rank and $\dot{w}(t) = \eta(t)\Psi(x, w)\tilde{e}(y) = 0$, then for $\eta(t) > 0, \tilde{e}(y)$ must be identically zero.

The authors' (incorrect) necessity proof :

> "For necessity, if the rank of $\Psi(x, \bar{w})$ is less than Pm, then $\exists \bar{w}$ such that $\dot{w}(t) = 0$ and $\tilde{e}(y) \neq 0$ (nontrivial solution); that is, the steady-state error is not an absolute minimum error $E[\bar{w}] = 0$. Therefore, $\Psi(x, \bar{w})$ must be of rank Pm."

does not preclude the trivial solution $\tilde{e}(y) = 0$ while the Rank $\Psi(x, \bar{w}) < Pm$ (Figure B-1). This is illustrated in the following counterexample.

Assume that in training a neural network we encounter a local minimum, $\bar{w}$, with $\tilde{e}(y) \neq 0$. The sufficiency condition states that $\Psi(x, \bar{w})$ must have deficient rank. Suppose now we define a new target set for the network which is identical to the current output of the network (i.e. at the local minimum). Then $\tilde{e}'(y) = 0$ (error of new target set) while $\Psi(x, \bar{w})$, which is unaffected by the change of target, remains deficient in rank. In other words, it is possible to have a target set and weight vector, $\bar{w}$, such that $\tilde{e}'(y) = 0$ while $\Psi(x, \bar{w})$ is rank deficient. This result is inconsistent with the theorem.

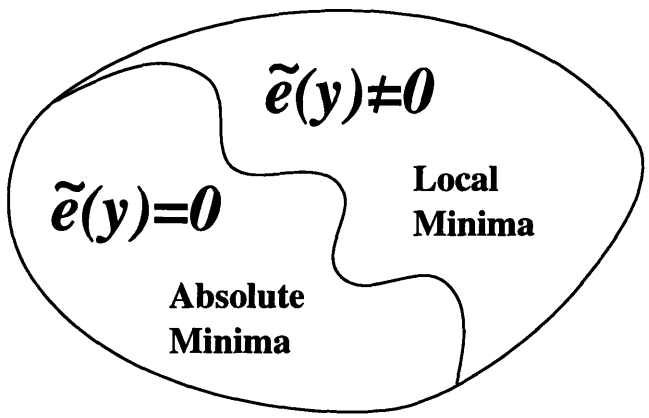


Figure B-1: Solution Set for Necessary Condition

Note that our counterexample assumes the existence of local minima. The validity of this assumption has been shown by Blum [11], who proved analytically the existence of local minima for the Exclusive-OR network.

Despite the incompleteness of the theorem, there is some empirical evidence that the matrix $\Psi(x, \bar{w})$ indeed has full rank at absolute minima for particular examples [65]. Unfortunately, the theorem as it stands, does not allow a definitive proof of the necessity of this condition in the general case.

Appendix C

Simulation Examples

This appendix describes the simulation examples used in this investigation.

C.1 Character Recognition

This example is a character recognition task. It was obtained from the University of California, Irvine repository for machine learning databases [49]. It was originally used in a study of Holland-style adaptive classifiers [20]. The results of this method were able to achieve just over 80 % accuracy in classification. There is no recorded usage of feedforward neural networks to learn this task.

The purpose of this task is to correctly classify characters based on 16 measurements of a raster image of the character. The measurements are both statistical moments and edge counts of the raw raster data. The description of the measurements is given in Table C.1. The patterns themselves consist of a letter output and 16 integer inputs. Each element of the input has a value between 0 and 15. The full data set contains 20000 characters. These were generated from 20 base fonts and random distortions thereof. The distribution of the characters in the full data set are given in Table C.2.

The simulations presented in this investigation uses only 2600 characters from the full data set, 100 for each letter. In addition, the inputs and output have been recoded. Each input element was coded into a string of 4 binary bits, therefore making

Name	Description	Data Type	Input/Output
lettr	capital letter	26 values from A to Z	output
x-box	horizontal position of box	integer	input
y-box	vertical position of box	integer	input
width	width of box	integer	input
high	height of box	integer	input
onpix	total # on pixels	integer	input
x-bar	mean x of on pixels in box	integer	input
y-bar	mean y of on pixels in box	integer	input
x2bar	mean x variance	integer	input
y2bar	mean y variance	integer	input
xybar	mean x y correlation	integer	input
x2ybr	mean of x * x * y	integer	input
xy2br	mean of x * y * y	integer	input
x-ege	mean edge count left to right	integer	input
xegvy	correlation of x-ege with y	integer	input
y-ege	mean edge count bottom to top	integer	input
yegvx	correlation of y-ege with x	integer	input

Table C.1: Attributes of Character Recognition Example

the input pattern 64 bits in total. Each character of the output was designated one output, therefore only one of 26 outputs would be active at any one time.

It is important to note that although this is not a very large example, it does represent a task where exact learning is required. As such, the size and accurate learning requirements combine to make this a useful problem to study.

Instances	Letter	Instances	Letter	Instances	Letter	Instances	Letter
789	A	766	B	736	C	805	D
768	E	775	F	773	G	734	H
755	I	747	J	739	K	761	L
792	M	783	N	753	O	803	P
783	Q	758	R	748	S	796	T
813	U	764	V	752	W	787	X
786	Y	734	Z				

Table C.2: Distribution of Examples in Character Recognition Example

C.2 Computer Speech

This example is designed to teach an artificial neural network to speak. The NETTalk data and network are described in detail in an investigation due to Sejnowski and Rosenberg [64]. The purpose of their investigation was to test the feasibility of teaching artificial neural networks to speak. In the process, the generalization and the data extraction capabilities of artificial feedforward neural networks are studied.

The input data is in the form of letters which are presented to the network in the context of other letter and other words. The output data is a phonetic coding of how the letter should be pronounced in this context, which can then be fed into a vocoder to listen to the speech.

Once trained, the artificial neural network was tested on other words for generalization ability, which the investigation characterized as good. In addition, the hidden units were studied to understand the features that were extracted from the input data.

The network used in the Sejnowski investigation had 203 (7×29) inputs, 80 hidden units and 26 output units. Approximately 1024 words were in the training set, and this yielded approximately 5400 training examples. The training time was on the order of days, as described in the investigation.

The large network size and long training time makes this a useful task on which to test parallel algorithms and hardware implementations. For example, the parallel algorithm on the Connection Machine CM-5 [78] used this task as a benchmark problem, and was able to take days of computation down to minutes. The CNAPS chips due to Hammerstrom [27], a hardware implementation of the MLP algorithm, was able to train the NETTalk network in 7 seconds [31].

This example represents a difficult problem in network training. Both the network size, training set size and required training time make this an interesting benchmark problem in the study of large artificial neural network tasks.

Appendix D

Computational Resources

This appendix outlines the computational resources used to produce the results in this investigation. All of the data presented in this investigation is a result of simulations run on the Adaptive Solution CNAPS Neurocomputer, however, the other platforms described here played an integral part in the development of concepts.

For each platform, the general architecture and its relationship to neural network computation is discussed. In addition, any neural network software used in this investigation is also described here.

This discussion is meant to demonstrate the wide range computational resources required to carry out the simulations described and by no means does it represent a complete description of the computational properties of each platform.

D.1 Adaptive Solutions CNAPS Neurocomputer

The Adaptive Solutions CNAPS Neurocomputer is a SIMD processing engine specifically designed for neural network applications [27]. Each processor consists of the basic elements of a simple add and multiply machine. The topology of the processors is in the form of a linear array with each processor connected to common IN and OUT buses and to left and right neighbours. If programmed efficiently, the machine is a powerful inner product engine, and as such performs excellently in the computation and training of Multilayer Perceptrons.

Each processing elements performs fixed point arithmetic and is limited to 16-bit signed weights and intermediate calculations and 8-bit inputs. As a result, architectures that are not robust to limited precision and dynamic range perform poorly e.g. the Two Layer Architecture described in Chapter 6.

Although this machine has multiple processors, the SIMD architecture precludes parallel training, except for very small applications. The Tree-Like Perceptron network was simulated in a serial fashion with the backpropagation program provided with the machine. The Tree-Like Shared network was simulated using a program adapted from the backpropagation program provided with the machine. The Two Layer network was simulated with the two hidden layer program provided with the machine.

An important computational consideration was that the author had exclusive access to this machine. As described below, the timesharing of even the most powerful computer can reduce its computational ability by orders of magnitude. As a result, all of the simulations described in this investigation have been run on this machine.

D.2 Connection Machine CM-5

The Thinking Machines Connection Machines CM-5 is a massively parallel computer developed for both MIMD and data parallel approaches to parallel computation. The processing elements are Sparc 10 RISC processors augmented with 8 floating point units. The topology of the network is highly flexible, but its native modes are NEWS mesh and torus.

Previous work on the CM-5 for parallel implementations of backpropagation is due to Zhang [78]. All the architecture simulators for this investigation were designed to take advantage of parallel training strategies. In addition, a simulator for multiple, parallel training sessions of MLP networks was designed to study the probabilistic nature of training.

Despite the computational power of the CM-5, the timesharing environment reduced it efficacy greatly, making it a highly flexible medium to small-scale simulator.

The delineation of large, medium and small-scale are somewhat arbitrary. It takes into account number of patterns to be learned, the size of the network and the time required to train it. Examples of large-scale problems are the NETTalk problem [64] example and the full data set of the Character Recognition example, both described in Appendix C. Medium-scale examples would be the scaled down version of the Character recognition problem used in this investigation (see Appendix C). Small-scale examples would range from XOR [62] to the SONAR problem [25].

D.3 CRAY X-MP

The CRAY X-MP is a powerful vector supercomputer. Although it is a multiprocessor machine, the CRAY X-MP is not a parallel computer. Thus results from the CRAY were performed in a serial training fashion. The advantage of the CRAY is its powerful vector computation engine. The vector computations make it especially suited for training the Multilayer Perceptron and similar network architectures.

The freeware ASPIRIN neural network simulator by Russell Leighton was used to perform calculations on the CRAY. It supports the use of vector libraries making it a powerful simulator. When compared to the workstation version of the software, described below, there was a 150 times speedup when run on the CRAY. This speedup also takes into account the timesharing environment of the CRAY. Although the CRAY has good computational performance, the CM-5 represented a much more flexible environment, especially for parallel training, thus the CRAY was used for some initial results and then phased out. Simulations that were originally run on the CRAY, were re-run on the Adaptive solutions machine with similar computational performance.

D.4 SUN Sparcstation 1

The SUN Sparc 1 workstation is a single processor workstation. As such, parallel training is impossible and serial training is slow. As a result, the Sparc workstation

was only used for small-scale testing.

The software used in small-scale testing was the ASPIRIN freeware package by Russell Leighton. This package was described under the description of the CRAY X-MP.

Bibliography

[1] A. Adams and S. J. Bye. New perceptron. *Electronic Letters*, 28(3), January 1992.

[2] Adaptive Solutions Incorporated, Beaverton, Oregon. *CNAPS Algorithm Series: Backpropagation Guide*, second edition, May 1993.

[3] B. Apolloni and G. Ronchini. Dynamic sizing of multilayer perceptrons. *Biological Cybernetics*, 71:49–63, 1994.

[4] Masahiko Arai. Bounds on the number of hidden units in binary-valued three-layer neural networks. *Neural Networks*, 6:855–860, 1993.

[5] Pierre Baldi and Kurt Hornik. Neural networks and principal component analysis : Learning from examples without local minima. *Neural Networks*, 2:53–58, 1989.

[6] Etienne Barnard. Optimization for training neural nets. *IEEE Transactions on Neural Networks*, 3(2), 1992.

[7] E.B. Baum. On the capabilities of multilayer perceptrons. *Journal of Complexity*, 4:193–215, 1988.

[8] E.B. Baum and D. Haussler. What size net gives valid generalization ? *Neural Computation*, 1:151–160, 1989.

[9] Adi Ben-Israel and Thomas N. E. Greville. *Generalized Inverses: Theory and Applications*. Robert E. Krieger Publishing Company, Huntington, New York, 1980.

[10] Friedrich Bielgler-König and Frank Bärmann. A learning algorithm for multilayered neural networks based on linear least squares problems. *Neural Networks*, 6:127–131, 1993.

[11] E. K. Blum. Approximation of boolean functions by sigmoidal networks : Part 1 - XOR and other two-variable functions. *Neural Computation*, 1:532–540, 1989.

[12] Martin L. Brady, Raghu Raghavan, and Joseph Slawny. Back propagation fails to separate where perceptrons succeed. *IEEE Transactions on Circuits and Systems*, 36(5):665–674, May 1989.

[13] David Cohn and Gerald Tesauro. How tight are the Vapnik-Chervonenkis bounds ? *Neural Computation*, 4:249–269, 1992.

[14] Morton L. Curtis. *Abstract Linear Algebra.* Springer-Verlag, New York, 1990.

[15] Jacques de Villiers and Etienne Barnard. Backpropagation neural nets with one and two hidden layers. *IEEE Transactions on Neural Networks*, 4(1), January 1992.

[16] Persi Diaconis and Mehrdad Shahshahani. On nonlinear functions of linear combinations. *Siam Journal of Scientific and Statistical Computation*, 5(1):175–191, 1984.

[17] Scott E. Fahlman. Faster-learning variations of back-propagation: an empirical study. In David Touretzky, Geoffrey Hinton, and Terrence Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School at Carnegie Mellon University*, pages 38–51, San Mateo, California, June 1988. Morgan Kaufmann.

[18] Françoise Fogelman Soulié and Jeanny Hérault, editors. *Neurocomputing : Algorithms, Architectures and Applications.* NATO Advanced Sciences Institute Series F : Computer and Systems Sciences. Springer-Verlag, Berlin, 1990.

[19] Marcus Frean. The Upstart algorithm: A method for constructing and training feedforward neural networks. *Neural Computation*, 2:198–209, 1990.

[20] P. W. Frey and D. J. Slate. Letter recognition using holland-style adaptive classifiers. *Machine Learning*, 6(2), March 1991.

[21] Yoshiji Fujimoto, Naoyuki Fukuda, and Toshio Akabane. Massively parallel architectures for large scale neural network simulations. *IEEE Transactions on Neural Networks*, 3(6):876–888, November 1992.

[22] Osamu Fujita. A method for designing the internal representation of neural networks and its application to network synthesis. *Neural Networks*, 4:827–837, 1991.

[23] Osamu Fujita. Optimization of the hidden units function in feedforward neural networks. *Neural Networks*, 5:755–764, 1992.

[24] Marco Gori and Alberto Tesi. On the problem of local minima in backpropagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(1), January 1992.

[25] R. P. Gorman and Sejnowski T. J. Analysis of hidden units in a layered network trained to classify sonar targets. *Neural Networks*, 1:75–89, 1988.

[26] Tal Grossman, Ronny Meir, and Eytan Domany. Learning by choice of internal representations. *Complex Systems*, 2:555–575, 1988.

[27] Dan Hammerstrom. A vlsi architecture for high-performance, low-cost, on-chip learning. *Proceedings of International Joint Conference on Neural Networks*, 2:537–544, 1990.

[28] Dan Hammerstrom. Neural networks at work. *IEEE Spectrum*, 30(6):26–32, June 1993.

[29] Dan Hammerstrom. Working with neural networks. *IEEE Spectrum*, pages 46–53, July 1993.

[30] John Hart Jr. and Barry Gordon. Neural subsystems for object knowledge. *Nature*, 359:60–64, September 1992.

[31] Simon Haykin. *Neural Networks : A Comprehensive Foundation.* Macmillan College Publishing Corporation, New York, New York, 1994.

[32] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the Theory of Neural Computation*, volume 1 of *Santa Fe Institute Studies in the Sciences of Complexity.* Addison-Wesley, Reading, Massachusetts, 1991.

[33] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4:251–257, 1991.

[34] Shih-Chi Huang and Yih-Fang Huang. Bounds on the number of hidden units in multilayer perceptrons. *IEEE Transactions on Neural Networks*, 2(1), January 1991.

[35] INTEL Corporation. *Real-time Pattern Recognition for Embedded Neural Networks.* INTEL 80170NX Electrically Trainable Analog Neural Networks (ETANN).

[36] Yoshio Izui and Alex Pentland. Analysis of neural networks with redundancy. *Neural Computation*, 2:226–238, 1990.

[37] Marwan Jabri and Barry Flower. Weight perturbation : An optimal architecture and learning technique for analog vlsi feedforward and recurrent multilayer networks. *Neural Computation*, 3:546–565, 1991.

[38] Qi Jia, Naohiro Toda, and Shiro Usui. A study on initial value setting of the back-propagation learning algorithm. *Systems and Computers in Japan*, 22(10), 1991.

[39] Thomas Kailath. *Linear Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1980.

[40] S. T. Kim, K. Suwunboriruksa, S. Herath, A. Jayasumana, and J. Herath. Algorithmic transformations for neural computing and performance of supervised

learning on a dataflow machine. *IEEE Transactions on Software Engineering*, 18(7):613–623, July 1992.

[41] Barbara J. Knowlton and Larry R. Squire. The learning of categories: Parallel brain systems for item memory and category knowledge. *Science*, 262:1747–1749, December 1993.

[42] John F. Kolen and Ashok K. Goel. Learning in parallel distributed processing networks : computational complexity and information content. *IEEE Transactions on Systems, Man and Cybernetics*, 21(2), 1991.

[43] John F. Kolen and Jordan B. Pollack. Backpropagation is sensitive to initial conditions. *Complex Systems*, 4:269–280, 1990.

[44] S. Y. Kung and J. N. Hwang. A unified systolic architecture for artificial neural network. *Journal of Parallel and Distributed Computing*, 6:358–387, 1989.

[45] Wei-Ming Lin, Victor K. Prasanna, and K. Wojtek Przytula. Algorithmic mapping of neural network models onto parallel simd machines. *IEEE Transaction on Computers*, 40(12), December 1991.

[46] Jerry B. Lont and Walter Guggenbühl. Analog CMOS implementation of a multilayer perceptron with nonlinear synapses. *IEEE Transactions on Neural Networks*, 3(3), September 1992.

[47] Marvin L. Minsky and Seymour A. Papert. *Perceptrons - An introduction to computational geometry.* The MIT Press, Cambridge, Massachusetts, expanded edition, 1988.

[48] Gagan Mirchandani and Wei Cao. On hidden nodes for neural nets. *IEEE Transactions on Circuits and Systems*, 36(5):661–664, May 1989.

[49] P. M. Murphy and D. W. Aha. UCI repository of machine learning databases. Machine-readable data repository, 1994. University of California Irvine, Department of Information and Computer Science.

[50] Alan P. Murray, Dante Del Corso, and Lionel Tarassenko. Pulse-stream vlsi neural networks mixing analog and digital techiniques. *IEEE Transactions on Neural Networks*, 2(2):193–204, March 1991.

[51] Paul Murtagh, Ah Chung Tsoi, and Neil Bergmann. A bit-serial systolic array implementation of a multilayer perceptron. *IEE Proceedings - E Computers and Digital Techniques*, 1992.

[52] Nils Nilsson. *The Mathematical Foundations of Learning Machines.* Morgan Kaufmann, San Mateo, California, 1990.

[53] Sang-Hoon Oh and Youngjik Lee. Effect of nonlinear transformations on correlation between weighted sums in multilayer perceptrons. *IEEE Transactions on Neural Networks*, 5:508–510, 1994.

[54] Erkki Oja. Principal components, minor components and linear neural networks. *Neural Networks*, 5:927–935, 1992.

[55] Sophocles J. Orfanidis. Gram-schmidt neural nets. *Neural Computation*, 2:116–126, 1990.

[56] D.-H. Park, B.-E. Jun, and J.-H. Kim. Novel fast training algorithm for multilayer feedforward neural networks. *Electronic Letters*, 26(6), 1992.

[57] Alain Petrowski, Gérard Dreyfus, and Claude Girault. Performance analysis of a pipelined backpropagation parallel algorithm. *IEEE Transactions on Neural Networks*, 4(6):970–981, November 1993.

[58] G. Qiu, M. R. Varley, and T. J. Terrell. Accelerated training of backpropagation networks by using adaptive momentum step. *Electronic Letters*, 28(4), February 1992.

[59] L. M. Reyneri and E. Filippi. Modified backpropagation algorithm for fast learning in neural networks. *Electronic Letters*, 26(19), 1990.

[60] Steve G. Romaniuk and Lawrence O. Hall. Divide and conquer neral networks. *Neural Networks*, 6:1105–1116, 1993.

[61] Jay G. Rueckl, Kyle R. Cave, and Stephen M. Kosslyn. Why are "what" and "where" processed by separate cortical systems ? a computational investigation. *Journal of Cognitive Neuroscience*, 1(2):171–186, 1989.

[62] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error propagation. In David E. Rumelhart, James L. McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing : Explorations in the Microstructure of Cognition. Volume 1 : Foundations*, volume 1, chapter 8. The MIT Press, Cambridge, Massachusetts, 1986.

[63] Robert Scalero and Nazif Tepedelenlioglu. A fast new algorithm for training feedforward neural networks. *IEEE Transactions on Signal Processing*, 40(1), 1992.

[64] Terrence J. Sejnowski and Charles R. Rosenberg. Parallel networks that learn to pronounce english text. *Complex Systems*, 1:145–168, 1987.

[65] Jagesh V. Shah and Chi-Sang Poon. Condition number as a convergence indicator in backpropagation. In *World Congress of Neural Networks - San Diego*, volume 3, pages 568–573, Hillsdale, New Jersey, May 1994. Lawrence Erlbaum Associates, Incorporated.

[66] Jagesh V. Shah and Chi-Sang Poon. Observations on "characterization of training errors in supervised learning using gradient-based rules". Neural Networks – Letter to the Editor, 1995. In Press.

[67] Sara A. Solla, Esther Levin, and Michael Fleisher. Accelerated learning in layered neural networks. *Complex Systems*, 2:625–640, 1988.

[68] Eduardo D. Sontag. On the recognition capabilities of feedforward nets. Technical report, SYCON - Rutgers Center for System and Control, April 1990. Report SYCON 90-03.

[69] Alessandro Sperduti and Antonina Starita. Speed up learning and network optimization with extended backpropagation. *Neural Networks*, 6:365–383, 1993.

[70] Haruhisa Takahashi, Etsuji Tomita, and Tsutomu Kawabata. Separability of internal representations in multilayer perceptrons with applications to learning. *Neural Networks*, 6(5):689–703, 1993.

[71] Gerald Tesauro. Scaling relationships in backpropagation learning : dependence on training set size. *Complex Systems*, 1:367–372, 1987.

[72] Gerald Tesauro, Yu He, and Subutai Ahmad. Asymptotic convergence of backpropagation. *Neural Computation*, 1:382–391, 1989.

[73] A. van Ooyen and B. Nienhuis. Improving the convergence of the backpropagation algorithm. *Neural Networks*, 5:465–471, 1992.

[74] Jun Wang and B. Malakooti. Characterization of training errors in supervised learning using gradient based rules. *Neural Networks*, 6(8):1073–1087, 1993.

[75] Andrew R. Webb and David Lowe. The optimised internal representaion of multilayer classifier networks performs nonlinear discriminant analysis. *Neural Networks*, 3:367–375, 1990.

[76] Paul Werbos. *Beyond regression: New tools for prediction and analysis in the Behavioural Sciences.* PhD thesis, Division of Applied Sciences, Harvard University, Cambridge, Massachusetts, 1974.

[77] Fraser A. W. Wilson, Séamas P. Ó Scalaidhe, and Patricia S. Goldman-Rakic. Dissociation of object and spatial processing domains in primate prefrontal cortex. *Science*, 260:1955–1958, June 1993.

[78] Xiru Zhang, Michael McKenna, Jill P. Mesirov, and David L. Waltz. The backpropagation algorithm on grid and hypercube architectures. *Parallel Computing*, 14:317–327, 1990.