A NEW METHOD FOR
CONSTRUCTING AND TRAINING
MULTILAYER NEURAL NETWORKS

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science
in Electrical Engineering

By

John O. Moody, BSEE

Panos J. Antsaklis, Director

Department of Electrical Engineering
Notre Dame, Indiana
April, 1993
A NEW METHOD FOR
CONSTRUCTING AND TRAINING
MULTILAYER NEURAL NETWORKS

Abstract

by

John Moody

Dependence identification, an algorithm for constructing multilayer neural networks, is presented in this thesis. Its distinctive features are that it transforms the training task into a series of quadratic optimization problems and it constructs an appropriate network architecture based on the training data. The quadratic optimization problems are solved by the block conjugate residual algorithm, an iterative Krylov subspace technique that efficiently solves multiple large systems of linear algebraic equations. The quadratic optimization/dependence identification algorithm extends previous work on quadratic optimization and significantly speeds up learning in feed-forward multilayer neural networks compared to standard backpropagation. The algorithm's speed and construction qualities make it a good candidate for real time applications. A number of examples are used to illustrate the speed and usefulness of the algorithm.
for my wife,

Tammy
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td></td>
<td>vi</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td></td>
<td>vii</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>ARTIFICIAL NEURAL NETWORKS</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>The Artificial Neuron</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>The Single Layer Neural Network</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>The Single Layer Neural Network Training Problem</td>
<td>6</td>
</tr>
<tr>
<td>2.4</td>
<td>The Multilayer Neural Network</td>
<td>8</td>
</tr>
<tr>
<td>2.5</td>
<td>Backpropagation</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>NETWORK CONSTRUCTION ALGORITHMS</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Continuous Networks</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Boolean Networks</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>QUADRATIC OPTIMIZATION</td>
<td>18</td>
</tr>
<tr>
<td>4.1</td>
<td>Error Analysis</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Error Cost Function Examples</td>
<td>23</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Unique Zero Error Solution</td>
<td>24</td>
</tr>
<tr>
<td>4.2.2</td>
<td>No Exact Solution</td>
<td>25</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Infinite Exact Solution</td>
<td>27</td>
</tr>
</tbody>
</table>
4.3 Multilayer Networks ........................................... 30

5 BLOCK CONJUGATE RESIDUAL .............................. 34
  5.1 Analysis of Error Convergence .......................... 36

6 DEPENDENCE IDENTIFICATION ............................... 42
  6.1 A Network Construction Example ..................... 43
  6.2 The Dependence Identification Algorithm ............ 46
  6.3 Increased Tolerance and Recomputed Weights .......... 49
  6.4 Discussion of Linear System Solutions ............ 50

7 EXAMPLES AND SIMULATIONS ............................... 53
  7.1 Dependence Identification vs. Backpropagation .. 54
    7.1.1 The XOR Problem .................................. 54
    7.1.2 Sine Wave Approximation .......................... 55
    7.1.3 2 Input/2 Output Continuous Function ........... 57
    7.1.4 3 Input/1 Output Continuous Function ........... 58
  7.2 Increased Tolerance and Recomputed Weights .......... 62
  7.3 Dependence Identification for Backpropagation Initial Conditions 64

8 CONCLUSIONS ................................................. 67

APPENDIX ...................................................... 70

A The Makenet Program ....................................... 70
  A.1 Input Format ....................................... 70
  A.2 Example Input File ................................. 73
  A.3 Source Code ....................................... 74

BIBLIOGRAPHY ................................................. 95
LIST OF TABLES

2.1 Commonly used neural activation functions .......................... 6
6.1 The XOR function .................................................. 43
7.1 Comparative results for the XOR problem. .......................... 54
7.2 Comparative results for the sine wave problem. ..................... 56
7.3 Comparative results for the 2 input/2 output sinusoidal function. 57
7.4 Comparative results for the 3 input/1 output system. ............... 60
7.5 Results of parity checker training ................................. 65
# LIST OF FIGURES

2.1 An artificial neuron. ...................................... 5  
2.2 The single layer neural network. .......................... 6  
2.3 A Multilayer neural network. .............................. 9  
4.1 Quadratic cost function with a unique zero error solution. ............. 25  
4.2 Nonlinear cost function with a unique zero error solution. ............. 26  
4.3 Quadratic cost function with no zero error solution. .................... 27  
4.4 Nonlinear cost function with no zero error solution. .................... 28  
4.5 Quadratic cost function with infinite zero error solutions. ............ 29  
4.6 Nonlinear cost function with infinite zero error solutions. ............ 30  
6.1 Components of the XOR neural network. ....................... 45  
7.1 Testing results of sine wave approximation. ....................... 56  
7.2 Testing results of 2 input/2 output sinusoidal function. ................ 59  
7.3 Testing results of 3 input/1 output system. .......................... 61  
7.4 Decrease of hidden layer neurons due to increased tolerance $\epsilon_p$. .... 63  
7.5 Overall network error vs. increased tolerance $\epsilon_p$. .................. 64  
7.6 Network error vs. BP iterations for the parity checker problem. ....... 66
I would like to thank my advisor Dr. Panos Antsaklis for the consistent guidance and thoughtful direction he has given me throughout my two years of graduate school. His helpful comments, suggestions, and clear view of the overall direction of my research have been invaluable in preparing and finishing this thesis.

I would also like to thank Dr. Michael Lemmon and Dr. John Uhran for participating in my thesis defense committee. Their insights and comments have improved this document and sharpened my analytical skills.

I owe thanks to the Arthur J. Schmitt Foundation for providing the fellowship that has made it financially possible for me to attend the University of Notre Dame.

Finally I would like to thank my wife Tammy for her help in proofreading as well as for providing the love, support, and understanding that have made my life in graduate school much easier than would have otherwise been possible.
Artificial neural networks are a relatively old field in the area of computer and computation science. Original work on the subject appeared well before the advent of the modern digital computer. Research on neural networks came to a near stand-still during the 60’s and 70’s, as attention was directed toward the new and exciting digital machines. Computers showed incredible potential and were made smaller, faster and more powerful every year. Computers are able to perform tasks such as numerical solution of equations, iterative algorithms, sorting and formal logic in tiny fractions of the time required for human beings. Computers were found to be excellent tools for solving problems described through algorithms. Human beings remained much more proficient in tasks such as language, generalization, intuition, association, and complex pattern recognition. It is for this reason that researchers in the field of pattern recognition turned their attention back to artificial neural networks in the early 1980’s. The new results in neural network theory prompted researchers in fields such as control to become involved in the study. It was hoped that through the study of neural networks some of these “characteristically biological” abilities could be imparted to machines [1],[3],[8],[18].

A neural network is a collection of simple nonlinear processing units (neurons) that, when arranged in a network of interconnections, exhibit complex behavior. The
idea is based on the function of actual biological neurons, though a biological neuron is definitely not a simple device. Biological neurons receive many different types of input, both electrical and chemical, and the full range of operations and uses for these operations are unknown. In fact, neural biologists study artificial neural networks in an attempt to infer how real neurons interact with each other. Fortunately even neurons based on very simple models can be used to perform surprisingly complex tasks, so their study has immediate practical value as well as being scientifically interesting.

The primary attributes of artificial neural networks are that 1) the architecture is parallel, leading to fast hardware implementations; 2) they have the ability to learn their functionality through training; and 3) they often are able to generalize the information for which they are trained in order to deal with situations that were not trained.

Feedforward neural networks are discussed in this thesis. A feedforward neural network is one in which the inputs propagate through one or more layers of neurons to the output. A single neuron may only affect neurons in the next consecutive layer (or the network output) and may not affect neurons in its own layer or in previous layers.

The dependence identification algorithm (DI) for constructing and training multilayer feedforward neural networks is introduced in this thesis. Dependence identification is an alternative and/or an enhancement to the backpropagation algorithm (BP). The dependence identification algorithm constructs neural networks, i.e., it determines the number of layers, the number of neurons within each layer, and the values of the weights. When networks are designed with backpropagation, the designer must guess at the number of layers and neurons within the network. Dependence identification is much faster than backpropagation training and is more systematic.
The major calculations performed by the dependence identification algorithm involve the minimization of multiple quadratic cost functions (quadratic optimization). These cost functions result from a change of variables performed on the original nonlinear neural network training problem. An analysis of the relationship between the original nonlinear problem and the new quadratic problem is presented in this thesis.

Quadratic optimization can be difficult and/or time consuming due to the size of the matrices involved as well as possible numerical instabilities and matrix conditioning problems. The block conjugate residual algorithm (BCR) is presented in this thesis as a stable, efficient, and fast method for performing quadratic optimization. An analysis of its error convergence shows that it will perform well for most problems.

An introduction to neural networks and neural network training are contained in the next chapter. Chapter 3 deals with methods for constructing neural networks. The single layer training method of quadratic optimization is described in chapter 4. The block conjugate residual algorithm is presented and analyzed in chapter 5. The dependence identification algorithm for multilayer neural network construction and training is introduced in chapter 6. Chapter 7 contains examples and simulations of dependence identification including comparisons to backpropagation. Concluding remarks are given in chapter 8. The appendix of this thesis contains the C source code for a program that constructs neural networks using the dependence identification algorithm.
CHAPTER 2

ARTIFICIAL NEURAL NETWORKS

This chapter provides an introduction to artificial neural networks and network training. Section 2.1 presents the artificial neuron. The single layer neural network and the single layer network training problem are discussed in sections 2.2 and 2.3. The multilayer neural network is introduced in section 2.4 and the backpropagation algorithm for multilayer neural network training is presented in section 2.5.

2.1 The Artificial Neuron

A single artificial neuron of the type commonly used in feedforward neural networks is depicted in figure 2.1. The neuron receives $m$ inputs, $(u_1, \ldots, u_m)$, and has a single output $y$. Each input $u_i$ is multiplied by a corresponding weight factor $w_i$. The output is formed by performing a nonlinear function $\{\phi : \mathbb{R} \rightarrow \mathbb{R}\}$, known as the neural activation function, on the sum of the weighted inputs, i.e.,

$$y = \phi(\sum_{i=1}^{m} w_i u_i)$$

The function $\phi$ is generally non-decreasing, nonlinear and bounded, though there are exceptions to each of these. Examples of commonly used neural activation functions appear in table 2.1.
2.2 The Single Layer Neural Network

A single layer neural network is shown in figure 2.2. It receives \( m \) inputs and produces \( n \) outputs with one artificial neuron for each output. The \( j^{\text{th}} \) output may be expressed as

\[
y_j = \phi\left(\sum_{i=1}^{m} w_{ij} u_i \right)
\]

where \( 1 \leq j \leq n \) and \( w_{ij} \) is the weight between the \( i^{\text{th}} \) input and the \( j^{\text{th}} \) output. The network may be expressed in matrix form by defining the inputs and outputs as vectors and the weights as a matrix. Then

\[
y = \Phi(uW)
\]

where

\[
y \in \mathbb{R}^{1 \times n} \quad u \in \mathbb{R}^{1 \times m} \quad W \in \mathbb{R}^{m \times n}
\]

and

\[
\{\Phi : \mathbb{R}^n \to \mathbb{R}^n\} \quad \Phi_i(x_i) = \phi(x_i)
\]
### Table 2.1: Commonly used neural activation functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>( \phi(x) = \begin{cases} 0 &amp; x &lt; 0 \ 1 &amp; x \geq 0 \end{cases} )</td>
</tr>
<tr>
<td>sign</td>
<td>( \phi(x) = \begin{cases} -1 &amp; x &lt; 0 \ 1 &amp; x \geq 0 \end{cases} )</td>
</tr>
<tr>
<td>sat</td>
<td>( \phi(x) = \begin{cases} -1 &amp; x &lt; -1 \ x &amp; -1 \leq x \leq 1 \ 1 &amp; x &gt; 1 \end{cases} )</td>
</tr>
<tr>
<td>tanh</td>
<td>( \phi(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} )</td>
</tr>
<tr>
<td>sigmoid</td>
<td>( \phi(x) = \frac{1}{1 + e^{-x}} )</td>
</tr>
<tr>
<td>linear</td>
<td>( \phi(x) = x )</td>
</tr>
<tr>
<td>gaussian</td>
<td>( \phi(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-c)^2}{2\sigma^2}} )</td>
</tr>
</tbody>
</table>

Figure 2.2: The single layer neural network.

### 2.3 The Single Layer Neural Network Training Problem

The overall network represents a static nonlinear mapping \( \mathbb{R}^m \to \mathbb{R}^n \) from the input to the output. The goal of neural network training is to make this mapping approximate, as close as possible, some desired mapping from \( \mathbb{R}^m \) to \( \mathbb{R}^n \). This process is referred to as training because the network is presented with input and output patterns from the desired mapping, and some kind of iterative technique is used to cause the output of the neural network to converge as close as possible to the desired responses.
Suppose there are \( p \) input and output patterns that are to be used to train a neural network, defined as follows:

\[
(u^k_i, d^k_j) \quad i = 1, \ldots, m \quad j = 1, \ldots, n \quad k = 1, \ldots, p
\]

where \( u^k_i \) is the \( i \)th input for the \( k \)th training pattern and \( d^k_j \) is the \( j \)th desired output for the \( k \)th training pattern. For the neural network to realize the desired input/output relationships perfectly, we would like to choose the \( w_{ij} \)'s such that

\[
y^k_j = \phi(\sum_{i=1}^{m} w_{ij} u^k_i) = d^k_j
\]

However it is quite possible that no solution exists for \( w_{ij} \) to exactly match the desired output, so the problem is reformulated as a minimization of the square of the errors between the network and desired outputs. The single layer neural network training problem is then to find a set of \( w_{ij} \)'s such that

\[
\min_{w_{ij}} \sum_{k=1}^{p} \sum_{j=1}^{n} (y^k_j - d^k_j)^2
\]

The problem can also be expressed in the matrix case:

\[
\min_W \text{trace} \left( (Y - D)^T (Y - D) \right) = \\
\min_W \text{trace} \left( (\Phi(UW) - D)^T (\Phi(UW) - D) \right)
\]

where

\[
Y \in \mathbb{R}^{p \times n} \quad U \in \mathbb{R}^{p \times m} \quad D \in \mathbb{R}^{p \times n}
\]

and

\[
\{ \Phi : \mathbb{R}^{p \times n} \to \mathbb{R}^{p \times n} \} \quad \Phi_{ij}(x_{ij}) = \phi(x_{ij})
\]

where \( 1 \leq i \leq p \), and \( 1 \leq j \leq n \). The matrix notation is useful for compactness of notation. Actual computations of equation (2.1) would be performed using scalars and summations rather than computing, storing, and multiplying entire matrices.
Several methods have been proposed to solve the training problem for single layer neural networks including steepest descent, least mean squares, and Rosenblatt’s *perceptron training algorithm* [8], [20]. A computationally efficient method for performing least mean squares training is proposed in chapter 5 by drawing parallels between equation (2.1) and the *block conjugate residual algorithm*.

### 2.4 The Multilayer Neural Network

A multilayer neural network with a single hidden layer is shown in figure 2.3. In this case the *m* inputs and the *n* outputs are separated by *l* “hidden layer” neurons. This layer of neurons is referred to as hidden because it does not explicitly appear in the input or output. The hidden layer keeps the input from directly appearing in the network’s output equation. An *h* layer network has *h* − 1 hidden layers and *h* weight matrices. Let *w*<sub>r,ij</sub> refer to the *i,j*<sup>th</sup> element of the weight matrix corresponding to the *r*<sup>th</sup> network layer, where 1 ≤ *r* ≤ *h*. Let *l*<sub>*r*</sub> refer to the number of neurons in the *r*<sup>th</sup> layer of network, and note that *l*<sub>*h*</sub> = *n*. The output of the *j*<sup>th</sup> neuron in the output layer of the network is then

\[
y_j = \phi(\sum_{i=1}^{l_{h-1}} w_{h,qj} \phi(\sum_{q=1}^{l_{h-2}} w_{h-1,q,p} \cdots \phi(\sum_{i=1}^{m} w_{1,i,q} u_i) \cdots))
\]

where 1 ≤ *j* ≤ *n* = *l*<sub>*h*</sub>. For a two layer network, as in figure 2.3, the output is

\[
y_j = \phi(\sum_{i=1}^{l_1} w_{2,qj} \phi(\sum_{i=1}^{m} w_{1,i,q} u_i))
\]

The entire network can be recursively expressed in matrix form by defining *H*<sub>*r*</sub> ∈ *R*<sup>1×lr</sup> as the output of the *r*<sup>th</sup> hidden layer.

\[
H_1 = \Phi(uW_1)
\]

\[
H_r = \Phi(H_{r-1}W_r) \quad \quad y = H_h = \Phi(H_{h-1}W_h)
\]

where

\[
y \in \mathbb{R}^{1 \times n} \quad u \in \mathbb{R}^{1 \times m} \quad W_r \in \mathbb{R}^{l_{r-1} \times l_r} \quad W_1 \in \mathbb{R}^{m \times l_1}
\]
Figure 2.3 shows the $m$th input as being a constant equal to 1. This is a common practice in that it allows the weights associated with input $u_m$ to act as biases for the hidden layer neurons.

Single layer neural networks may only realize functions where the input/output space is linearly separable (though the input/output mapping may still be nonlinear). Multilayer neural networks are important because they can approximate arbitrary functions with any desired degree of accuracy as the number of hidden layer neurons approaches infinity; see [5], [10], and [12]. Two layer neural networks are sufficient for approximating smooth continuous functions, and three layer networks can handle discontinuities [4]. The goal in training a multilayer neural network is still the same, i.e., we wish to minimize the trace of $((Y - D)^T(Y - D))$ where $Y$ and $D$ are the same as defined in section 2.3 for the single layer neural network. However now there are multiple weight matrices and multiple levels of nonlinear activations. Thus the training goal is

$$
\min_{W} \text{trace} \left( (\Phi \ldots (\Phi(UW_1) \ldots W_h) - D)^T(\Phi \ldots (\Phi(UW_1) \ldots W_h) - D) \right) \quad (2.2)
$$
Methods used to train single layer neural networks do not, in general, work for the multilayer case. Single layer training methods normally rely on knowing the desired outputs of the network, but the desired outputs of the hidden layer neurons are unknown in the multilayer case.\(^{1}\)

### 2.5 Backpropagation

The most popular method of solving the multilayer neural network training problem (2.2) is known as backpropagation, developed by Rumelhart et. al. [21] The method involves performing a gradient descent of the error cost function. It is known as backpropagation because first the input is passed forward through the network to form an error value, and then the error is fed backward through the network to update the weight values.

The error cost function is defined as

\[
E = \text{trace} \left((Y - D)^T(Y - D)\right)
\]

which is rewritten in terms of scalars and expanded to include the weights

\[
E = \sum_{k=1}^{p} \sum_{j=1}^{n} (y_j^k - d_j^k)^2
\]

\[
= \sum_{k=1}^{p} \sum_{j=1}^{n} (\phi(\sum_{q=1}^{l_{k-1}} w_{k,q;ij}) \cdots \phi(\sum_{i=1}^{m} w_{1,qk-1;ij} u_i^k) \cdots) - d_j^k)^2
\]

(2.3)

where \(w_{r,ij}\) is the \(i, j^{th}\) element of weight matrix for the \(r^{th}\) network layer, \(u_i^k\) is the network input, \(y_j^k\) is the network output, and \(d_j^k\) is the desired output, all for the \(k^{th}\) training pattern. Partial derivatives of \(E\) with respect to the weights are taken in order to determine the gradient:

\[
\frac{\partial E}{\partial w_{r,ij}} = -2 \sum_{k=1}^{p} \delta_{r,j}^k \lambda_{r-1,i}^k
\]

(2.4)

\(^{1}\)This is another reason for the use of the term “hidden.”
where $\lambda_{r-1,i}^k$ refers to the output of the $i^{th}$ hidden layer neuron in the $(r - 1)^{th}$ layer of the network to the $k^{th}$ training pattern. Let $\lambda_{0,i}^k = u_i^k$, the $i^{th}$ network input for the $k^{th}$ pattern and $l_0 = m$, the number of inputs. The $\delta$ values are given by

$$
\delta^k_{r,j} = \begin{cases} 
    (d_j^k - y_j^k)\phi'(\sum_{i=1}^{l_{r-1}} w_{h,ij} \lambda_{h-1,i}^k) & \text{for } r = h \text{ (Output Layer)} \\
    (\sum_{q=1}^{l_{r+1}} \delta_{r+1,q}^k w_{r+1,jq})\phi'(\sum_{i=1}^{l_r} w_{r,ij} \lambda_{r-1,i}^k) & \text{for } r < h
\end{cases}
$$

where $\phi'(x) = \frac{d\phi(x)}{dx}$ the first derivative of the neural activation function. The $\delta$ values for the $r^{th}$ layer are created by multiplying the $\delta$ values for the next layer $(r + 1)$ by the weights of the next layer and by the derivatives of the input values of the $r^{th}$ layer. With this definition, the $\delta$ value for the “$(h + 1)$th layer” is simply the network error $(d_j^h - y_j^h)$. The other $\delta$ values are then found by working backward. This is the origin of the term error backpropagation.

The network weights are iteratively updated in the negative direction of the gradient. The distance traveled along the gradient at each iteration is defined by the “learning rate” $\eta$, which normally absorbs the factor of 2 that appears in equation (2.4). In order to help keep the solution from falling into local minima of the cost space, a “momentum term” with constant $\alpha$ is also added to the weight change rule. The weight change rule for the $\mu^{th}$ iteration of backpropagation is given by

$$
w_{r,ij}(\mu) = -\eta \frac{\partial E}{\partial w_{r,ij}} + \alpha w_{r,ij}(\mu - 1)
$$

where $\eta, \alpha \in \mathbb{R}, \eta > 0, \alpha \geq 0$. $w_{r,ij}(\mu)$ refers to the values of the weights in the $r^{th}$ layer at the $\mu^{th}$ iteration of backpropagation.

Backpropagation has numerous problems due to its being a gradient descent algorithm. It is very slow to converge and extremely sensitive to the magnitude of $\eta$ and $\alpha$. Choosing $\eta$ or $\alpha$ too high can lead to numeric instabilities and solutions that never converge. Choosing them too small (especially $\eta$) can make the solution time agonizingly long. Even when using the momentum term ($\alpha > 0$) the algorithm is
prone to falling into local minima of the cost function, making it very sensitive to the initial conditions of the weights.

Note that equation (2.5) uses the derivative of the neural activation function $\phi$. Standard backpropagation may fail if the activation function contains discontinuities or contains truly flat regions with zero derivatives. For example, the step and threshold functions have zero derivatives everywhere except for the point where the step is made. If an attempt is made to backpropagate an error in one of the flat regions, then the corresponding value of $\delta$ will become zero and no information will be returned. The problem is worse at the point of discontinuity because here the activation's derivative is not even defined. This limits backpropagation from working with many of the functions from table 2.1.

With all of these problems associated with backpropagation, the reader may wonder why it is so popular. The answer is, despite its many problems, because it works. Finding a good scheme for solving equation (2.2) is quite difficult. In fact the lack of a method for solving the multilayer neural network training problem was a prime factor in the near death of neural network research during its early days. Much research has gone into finding ways of improving backpropagation, and some of these methods represent considerable improvement, but all of them are still based on the same gradient descent which will always have the problems discussed above to some degree or another.
CHAPTER 3

NETWORK CONSTRUCTION ALGORITHMS

In the recent past, much work has gone into finding alternatives or improvements to backpropagation for training and constructing multilayer neural networks. Work which is relevant to this thesis will be detailed in the next two chapters. This chapter will detail some algorithms and methods for actually constructing neural networks, as well as training them.

Recall from section 2.5 that backpropagation must be given the number of hidden layers as well as the number of neurons within each layer. Unfortunately it is difficult to determine beforehand how many hidden layer neurons are necessary, much less optimal. Because of this problem, a considerable amount of work has gone into discovering methods of constructing neural networks based on the I/O patterns for which they are to be trained. The work on this subject can be divided into methods for continuous and binary valued network construction and are dealt with separately in the following two sections.

3.1 Continuous Networks

The work on network construction algorithms for continuous valued neural networks has concentrated on pruning or adding techniques that are used with the backpropagation training algorithm (see section 2.5). With adding techniques, the
network starts out very small and new neurons are added when the error minimization reaches some unacceptable minimum. Conversely pruning techniques start out with a very large network, with the goal being to eliminate rarely used weights and/or neurons.

The major tool for pruning weights and neurons is a method known as “weight decay” summarized in [8] based on work presented in [9], [15], and [24]. The point of weight decay is to have all of the weights in the neural network decrease slightly in magnitude at every backpropagation training step. If BP training is constantly emphasizing certain weights, then the slight decay will not hurt these values, however if a weight is rarely changed by BP, then the decay process will make it go to zero. It is then possible to remove the weights which have near zero magnitude.

As a first try, weight decay can be accomplished with the following equation:

\[ w_{ij}^{\text{new}} = (1 - \epsilon) w_{ij}^{\text{old}} \]  
(3.1)

where \( w_{ij}^{\text{new}} \) is the new value of the \( ij \)th weight and \( \epsilon \) is a constant decay factor. This rule causes the BP cost function to take on the following form:

\[ E = E_0 + \frac{1}{2} (1 - \epsilon) \sum_{ij} w_{ij}^2 \]  
(3.2)

where \( E_0 \) is the original backpropagation cost function, equation (2.3). Equation (3.2) shows that the cost function is penalizing high weight values, since the square of the weights appears directly as part of the cost function. The intended purpose of equation (3.1) is to erode weights which are not constantly reinforced, not to penalize high weights. Redefining the cost function as

\[ E = E_0 + \frac{1}{2} (1 - \epsilon) \sum_{ij} \frac{w_{ij}^2}{1 + w_{ij}^2} \]  
(3.3)

gives rise to the following form of weight decay

\[ w_{ij}^{\text{new}} = (1 - \epsilon_{ij}) w_{ij}^{\text{old}} \]  
(3.4)
where
\[ \epsilon_{ij} = \frac{\epsilon_0}{(1 + w_{ij}^2)^2} \] (3.5)
and \( \epsilon_0 \) is a constant. This new rule allows the weight decay for a particular weight to be scaled according to the weight’s current magnitude and does not penalize large weights.

So far the rules have concentrated on individual weights rather than on neurons. It is possible to modify equation (3.5) to cover all of the weights that enter the \( i^{th} \) neuron by defining
\[ \epsilon_i = \frac{\epsilon_0}{(1 + \sum_j w_{ij}^2)^2} \] (3.6)
Then all of the weights connected to the \( i^{th} \) neuron could be decayed using equation (3.4) and replacing \( \epsilon_{ij} \) with \( \epsilon_i \). In this way entire neurons may be removed from the network.

The method of weight decay has two main problems, it requires that the network be very large at the beginning of training and it relies on backpropagation. Hardware or time restrictions may interfere with the size requirements for the initial network, and backpropagation has its own host of problems (see section 2.5).

### 3.2 Boolean Networks

The most impressive progress in network construction algorithms has been for boolean networks. Boolean networks realize boolean functions, with the inputs, outputs and neural activations all being binary valued. Three of these algorithms will be discussed here, as presented in [8]. The networks will be discussed as having many inputs and only one output, but it is not difficult to extend these results to the multi-output cases. In all of the boolean network construction algorithms, the hidden or auxiliary neurons are used to linearly separate the pattern space.

Marchand et al. [16] developed a method for constructing a boolean network with a single hidden layer. The inputs and outputs and hidden layer activations are all
±1. The idea is to train a single hidden layer neuron to give a correct response for all of the patterns which should generate a +1 output and at least one of the patterns that generates a -1 response. New hidden units are then added, each one getting one more of the -1 responses correct, while maintaining the correct response for all of the +1 patterns. When all of the -1 responses are answered correctly by at least one hidden layer unit, Marchand et al. prove that the perceptron training algorithm [20] can be applied to produce the output layer weights, i.e., they prove that the hidden layer linearly separates the pattern space.

The upstart algorithm developed by Frean in 1990 [7] uses a different kind of network architecture. The first step is to train a single neuron to correctly classify a portion of the patterns. If not all of the patterns are correctly classified, then two new neurons are added to the network. These neurons receive the input patterns as their own input and feed their output to the first neuron. One of these neurons has a positive weight and the other has a negative weight. The two neurons' weights are then used to correct some of the mistakes made by the first neuron. If the first neuron still does not classify all patterns correctly then four new neurons are added, two for each of the secondary neurons, and the process is repeated until the first neuron correctly classifies all of the I/O patterns. Frean proves that each additional layer will eliminate at least one of the incorrectly classified patterns at the output layer, thus the process must eventually halt.

The tiling algorithm was developed by Mézard and Nadal [17] in 1989. The algorithm is based on the idea that if two input patterns are to give rise to different output values, then their hidden layer representations must also be different. First a single neuron is trained to correctly classify as large a proportion of the pattern space as possible. Extra units are then added until no two input patterns which have opposite output values have the same hidden layer representation. The process is then restarted, and the hidden layer patterns are used as inputs to a new layer.
Mézard and Nadal prove that a layer will always have at least one fewer neuron than the layer before it, thus the algorithm will always reach a conclusion with a single output that gives the correct response for all patterns.
CHAPTER 4
QUADRATIC OPTIMIZATION

The single layer neural network training goal (see section 2.3) is to find a weight matrix \( W \) that

\[
\min_W \text{trace } ((D - Y)^T (D - Y)) = \text{trace } ((D - \Phi(UW))^T (D - \Phi(UW)))
\]

where \( U \) is the matrix of input patterns, \( Y \) is the matrix of network outputs to the input patterns, \( D \) is the matrix of desired output patterns, and \( \phi \) is the (nonlinear) neural activation function. In [22], Sartori and Antsaklis provide a novel method for dealing with the nonlinearity is proposed. Consider defining a new matrix \( V \in \mathbb{R}^{p \times n} \) such that

\[ \Phi(V) = D \]

(4.1)

that is,

\[ \phi(v_{ij}) = d_{ij} \]

for \( 1 \leq i \leq p, 1 \leq j \leq n \) and \( v_{ij} \) and \( d_{ij} \) are the \( ij^{th} \) elements of \( V \) and \( D \) respectively. If \( \phi \) has a well defined inverse, then we can take \( V = \Phi^{-1}(D) \), however it is not necessary that \( \phi \) be bijective. For example, the step function is surjective for the domain of \( \mathbb{R} \) and the range of \( \{0, 1\} \), i.e., it has an infinite number of inputs.
which will evaluate to 1 and an infinite number of inputs which will evaluate to 0. The elements of $V$ can still be defined by making a projection of the inverse image of $\phi$, i.e., assign some positive number to $v_{ij}$ whenever $d_{ij}$ is 1 and assign some negative number to $v_{ij}$ whenever $d_{ij}$ is 0. The numbers assigned may be random or some pre-chosen constants. With the change of variables in effect, the single layer neural network training problem reduces to

$$\min_W \hat{F}(W)$$

where

$$\hat{F}(W) = \text{trace } ((V - UW)^T (V - UW))$$

Expanding equation (4.3) yields

$$\hat{F}(W) = \text{trace } (W^T U^T U W - 2W^T U^T V + V^T V)$$

In order to minimize $\hat{F}(W)$ with respect to $W$, we take its derivative and set it equal to zero:

$$\frac{d\hat{F}}{dW} = 2U^T UW - 2U^T V = 0$$

Thus to solve the problem (4.2) we must solve

$$U^T UW = U^T V$$

(4.5)

If $U^T U$ is positive definite (full rank) then

$$W = (U^T U)^{-1} U^T V$$

Even if $U^T U$ is not full rank, equation (4.5) may be solved using least means squares methods to obtain the solution to problem (4.2).
4.1 Error Analysis

The original single layer neural network training problem (2.1) is to find a $W$ which

$$\min_W F(W)$$

(4.6)

where

$$F(W) = \text{trace} \ (D - \Phi(UW))^T (D - \Phi(UW))$$

Let the error matrix, $\epsilon \in \mathbb{R}^{p \times n}$, associated with equation (4.6) at any time be given by

$$\epsilon = D - \Phi(UW)$$

(4.7)

For the case where the change of variables, $\Phi(V) = D$, has been performed, the problem is defined by equations (4.2) and (4.3). The error matrix, $\hat{\epsilon} \in \mathbb{R}^{p \times n}$, associated with problem (4.2) at any time is given by

$$\hat{\epsilon} = V - UW$$

(4.8)

The relation between $\epsilon$ and $\hat{\epsilon}$ is given in [22] as

$$\epsilon = \Phi(UW + \hat{\epsilon}) - \Phi(UW)$$

(4.9)

Note that each element of $\epsilon$ depends only on the corresponding element of $\hat{\epsilon}$ not on the entire $\hat{\epsilon}$ matrix, i.e., $\epsilon^k_j$ is a function of $\hat{\epsilon}^k_j$ only:

$$\epsilon^k_j = \phi((\sum_{i=1}^{m} u^k_i w_{ij}) + \hat{\epsilon}^k_j) - \phi(\sum_{i=1}^{m} u^k_i w_{ij})$$

where $1 \leq j \leq n$ and $1 \leq k \leq p$.

An examination of equation (4.9) shows that for a given $W$, if $\hat{\epsilon} = 0$, then $\epsilon = 0$ as well. This is true for any neural activation function, sigmoid, hyperbolic tangent, signum, step, etc.... Thus if the new problem has an exact solution, then an exact solution to the original problem has been found as well. If the neural activation
function is bijective, e.g. the sigmoid, then for any \( W, \epsilon = 0 \) if and only if \( \hat{\epsilon} = 0 \). In general however \( \hat{\epsilon} \neq 0 \).

It is interesting to express the gradient and the optimal value of \( \hat{F}(W) \) in terms of error \( \hat{\epsilon} \). Combining equations (4.4) and (4.8) shows the gradient of \( \hat{F}(W) \) is given by

\[
\nabla \hat{F}(W) = -2U^T \hat{\epsilon}
\]

(4.10)

A zero gradient will occur if \( \hat{\epsilon} = 0 \) or if \( U^T \hat{\epsilon} = 0 \). The \( W \) which minimizes \( \hat{F}(W) \) (makes the gradient equal to zero) is obtained from equation (4.5). Let this minimizing solution, call it \( W^* \) be the solution of \( U^T U W^* = U^T V \). At this minimum, the cost function is equal to

\[
\hat{F}(W) = \text{trace} (W^{*T} U^T U W^* - 2W^{*T} U^T V + V^T V)
\]

\[
= \text{trace} (U^T U W^* - 2W^{*T} U^T U W^* + V^T V)
\]

\[
= \text{trace} (V^T V - W^{*T} U^T U W^*)
\]

\[
= \text{trace} (V^T (V - U W^*))
\]

Thus

\[
\hat{F}(W) = \text{trace} V^T \hat{\epsilon}^*
\]

\[
\hat{\epsilon}^* = V - U W^*
\]

(4.11)

Similarly, the gradient of nonlinear cost function, \( F(W) \), is given by

\[
\nabla F(W) = -2U^T \text{diag} \left( \nabla (\Phi(U W)) \right) \epsilon
\]

(4.12)

assuming that the activation function is differentiable, which is the case with the sigmoids. If \( \nabla F(W) = 0 \) then \( W \) is a minimum of \( F(W) \). This zero gradient can occur if \( \epsilon = 0 \) or if \(-2U^T \text{diag} \left( \nabla (\Phi(U W)) \right) \epsilon = 0 \), which can be rewritten as

\[
\sum_{k=1}^{p} u_i^k \frac{\partial \Phi(u_i^k w_{ij})}{\partial (u_i^k w_{ij})} \hat{\epsilon}_j^k = 0
\]
for all $i$ and $j$.

If $\dot{e} = 0$, then $\epsilon$ is also equal to zero as shown above. If $\dot{e} \neq 0$ and $U^T \dot{e} = 0$ then $W$ is a minimum of $\tilde{F}(W)$ but not necessarily a minimum of $F(W)$. If, in addition,

$$\frac{\partial \phi(u^k w_{ij})}{\partial (u^k w_{ij})} \approx 1 \quad (4.13)$$

then $W$ is very close to a minimum of $F(W)$. This can happen if the magnitudes of $u^k w_{ij}$ are very large, pushing the outputs into the "flat" regions of certain neural activation functions such as the sigmoid and hyperbolic tangent.

The role of the flat region of the sigmoid can also be seen by using the following approximation for $\epsilon$

$$\epsilon^k_j \approx \epsilon^k_j \phi'(\sum_{i=1}^{m} u^k_i w_{ij}) \quad (4.14)$$

where $\phi'(x) = \frac{d\phi}{dx}(x)$. The approximation is valid because

$$\phi'(\sum_{i=1}^{m} u^k_i w_{ij}) = \lim_{\epsilon^k_j \to 0} \frac{\phi((\sum_{i=1}^{m} u^k_i w_{ij}) + \epsilon^k_j) - \phi(\sum_{i=1}^{m} u^k_i w_{ij})}{\epsilon^k_j}$$

where $1 \leq j \leq n$ and $1 \leq k \leq p$.

It is now of interest to examine the conditions under which the $W$ which minimizes $\tilde{F}(W)$ adequately approximates the minimum of the original (nonlinear) problem. Suppose we find a weight matrix $W^*$ which solves problem (4.2). The associated error matrix is then

$$\epsilon^* = V - UW^*$$

Under what conditions are the elements of $\epsilon^*$, the nonlinear problem’s error matrix from equation (4.7) with weight matrix $W^*$, less than or equal to the elements of $\epsilon$ where $\epsilon$ is the error matrix associated with any weight matrix? Using the approximation, equation (4.14), we want

$$|\epsilon^k_j | \approx |\epsilon^k_j \phi'(\sum_{i=1}^{m} u^k_i w_{ij})| \leq |\epsilon^k_j \phi'(\sum_{i=1}^{m} u^k_i w_{ij})| \approx |\epsilon^k_j|$$
where \(w^*_{ij}\) and \(w_{ij}\) are the \(ij\)th elements of \(W^*\) and \(W\). The inequality should hold for all patterns \(k\) and output neurons \(j\). The inequality can be viewed in terms of the ratio of the slopes of the output activations compared to the ratio of the magnitudes of the errors:

\[
\frac{\phi'\left(\sum_{i=1}^{m} u_k^i w^*_{ij}\right)}{\phi'\left(\sum_{i=1}^{m} u_k^i w_{ij}\right)} \leq \frac{\hat{e}_j^k}{\hat{e}_j^k^*}
\]

(4.15)

Let \(\alpha\) equal the absolute value of the ratio of the slopes:

\[
\alpha = \left|\frac{\phi'\left(\sum_{i=1}^{m} u_k^i w^*_{ij}\right)}{\phi'\left(\sum_{i=1}^{m} u_k^i w_{ij}\right)}\right|
\]

Then for the weight matrix \(W^*\) to produce a minimum in the nonlinear space we need

\[
|\hat{e}_j^k^*| \leq \frac{1}{\alpha} |\hat{e}_j^k|
\]

If \(\hat{e}^* = V - UW^* = 0\) then the inequality will obviously hold. If the elements of \(UW^*\) are high in magnitude, then they will lie along the flat points of the sigmoid and the derivative will be approximately zero, making \(\alpha \approx 0\). In this case the solution \(W^*\) should lie close to the optimal solution of the nonlinear problem. If the elements of \(UW^*\) are small in magnitude, then the derivative of the sigmoid will be high and it is likely that \(W^*\) lies farther away from the optimal solution to the nonlinear problem. Note that the elements of \(W^*\) are not considered by themselves. It is necessary to examine the elements of \(UW^*\). This indicates that the set of training patterns has a strong impact on the value of the solutions that are derived.

### 4.2 Error Cost Function Examples

Consider a single layer two input one output neural network with \(\phi(x) = \tanh(x)\). The network is capable of realizing functions of the following form exactly:

\[
y = \tanh(u_1 w_1 + u_2 w_2)
\]

The following examples will explore the differences between the nonlinear and quadratic cost functions when the network is trained to approximate a function that can
be realized exactly (section 4.2.1), that has no exact solution (section 4.2.2), and that has infinite solutions (section 4.2.3).

### 4.2.1 Unique Zero Error Solution

The network is to be trained to realize the following function:

\[ d = \tanh(2u_1 + 3u_2) \]

which has an exact solution at \( W^* = \begin{bmatrix} 2 & 3 \end{bmatrix}^T \). The following ten pattern \( U \) and \( D \) matrices are created:

\[
U = \begin{bmatrix}
-1.4052 & 0.1485 \\
-2.2648 & 0.8557 \\
0.8943 & -2.4615 \\
0.8965 & -0.5829 \\
2.1735 & -2.1658 \\
-0.5825 & -0.4126 \\
0.0971 & 0.9339 \\
1.6548 & 0.4449 \\
-2.3271 & 2.1522 \\
-2.2327 & 1.7308 \\
\end{bmatrix}
\]

\[
D = \begin{bmatrix}
-0.9825 \\
-0.9613 \\
-1.0000 \\
0.0443 \\
-0.9732 \\
-0.9838 \\
0.9950 \\
0.9998 \\
0.9471 \\
0.6212 \\
\end{bmatrix}
\]

(4.16)

Figure 4.1 shows the error described by the quadratic cost function equation (4.2). The "*" on the graph at coordinates \((2, 3)\) represents the global minimum that corresponds to \( \hat{\epsilon} = 0 \) with solution \( W^* \). The previous section demonstrated when \( \hat{\epsilon} = 0 \) then the solution \( W^* \) will also cause \( \epsilon = 0 \). This is indeed the case as can be seen in figure 4.2 which shows the nonlinear error cost function described by equation (2.1). Notice the highly nonlinear nature of the surface shown in figure 4.2. This demonstrates that even with extremely simple networks, the actual nonlinear cost function can be quite complicated.
4.2.2 No Exact Solution

In this example, the network can not realize the desired function exactly, but can only approximate it. The desired function is

\[ d = 0.9 \tanh(2u_1 + 3u_2) \]

The same \( U \) matrix defined in equation (4.16) is used and the corresponding new \( D \) matrix is created. Figure 4.3 shows the quadratic error function. The minimum\(^1\) is given by \( \hat{W}^* = \begin{bmatrix} 0.75 & 1.25 \end{bmatrix}^T \). This minimum is marked by a ‘Q’ on the graph. The minimum for the actual (nonlinear) problem is at \( W = \begin{bmatrix} 1.25 & 2 \end{bmatrix}^T \) and is marked with an ‘N’ on the graph. The nonlinear cost function is shown in figure 4.4 with the two minima marked as above.

The derivative values corresponding to the solution of the quadratic function are

\(^1\)The weights in these examples are only computed at intervals of 0.25, thus the actual minimizing weights for the two cost functions may be slightly different than the values given.
Figure 4.2: Nonlinear cost function with a unique zero error solution.

given by

\[
\Phi'(U\hat{W}^*) = \begin{bmatrix}
0.5093 \\
0.6894 \\
0.0320 \\
0.9968 \\
0.3725 \\
0.4510 \\
0.2851 \\
0.1041 \\
0.4561 \\
0.7944
\end{bmatrix}
\]

which are not in general near zero, and indicate why the solution to the quadratic problem is not nearer to the solution of the nonlinear problem.
4.2.3 Infinite Exact Solution

In this example, there are an infinite number of possible solutions to the problem. The desired function is again

\[ d = \tanh(2u_1 + 3u_2) \]  

(4.17)
Figure 4.4: Nonlinear cost function with no zero error solution. Q = quadratic minimum, N = Nonlinear minimum.

However now the $U$ matrix is changed such that both columns are identical. The $U$ and $D$ matrices are now

$$U = \begin{bmatrix} -1.4052 & -1.4052 \\ -2.2648 & -2.2648 \\ 0.8943 & 0.8943 \\ 0.8965 & 0.8965 \\ 2.1735 & 2.1735 \\ -0.5825 & -0.5825 \\ 0.0971 & 0.0971 \\ 1.6548 & 1.6548 \\ -2.3271 & -2.3271 \\ -2.2327 & -2.2327 \end{bmatrix}$$

$$D = \begin{bmatrix} -1.0000 \\ -1.0000 \\ 0.9997 \\ 0.9997 \\ 1.0000 \\ -0.9941 \\ 0.4506 \\ 1.0000 \\ -1.0000 \\ -1.0000 \end{bmatrix}$$

This network has infinite solutions, along the line $w_1 + w_2 = 5$. Figures 4.5 and 4.6 show the quadratic and nonlinear cost functions respectively. The quadratic cost function has the same line of zero errors as the the nonlinear cost.
The block conjugate residual algorithm (discussed in chapter 5) finds the weight matrix $\hat{W}^* = \begin{bmatrix} 2.5 & 2.5 \end{bmatrix}^T$ given the quadratic cost function and an initial guess of $\hat{W} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$. This is a valid solution to the problem for the particular matrix $U$, but note that the desired function defined at the beginning of this section has an associated weight matrix of $W = \begin{bmatrix} 2 & 3 \end{bmatrix}^T$ (equation (4.17)). If the desired function were really $d = \tanh(2u_1 + 3u_2)$ and $u_1$, and $u_2$ were allowed to have different values, then we would have found an incorrect solution, not because of some flaw with the training method but because of an unwise choice of $U$.

The examples in the past three sections have shown that the method of quadratic optimization will solve the single layer neural network training problem exactly if at least one exact solution exists. This is even the case when the nonlinear error cost function is highly irregular or has large flat regions that would hinder gradient descent techniques. The example in section 4.2.2 shows that the when an exact solution to the problem does not exist, then quadratic optimization can come close to the optimal solution, but it will probably not achieve it exactly. Section 4.2.3
demonstrated that care must be taken when choosing the set of training patterns so that the training data accurately reflects the kinds of inputs the trained network will receive. This situation is not unique to quadratic optimization but is present for all training algorithms.

4.3 Multilayer Networks

The problem with using the change of variables from quadratic optimization to solve multilayer networks is that the desired activations of the hidden layer neurons are not known. In order to solve this problem, Sartori and Antsaklis [22] propose a method which uses the rules of backpropagation to produce a heuristic for estimating the desired hidden layer activations.

The equations for backpropagation are reprinted from section 2.5 for clarity. The error function which is being minimized is given by

$$E = \text{trace} \left( (Y - D)^T (Y - D) \right)$$

Partial derivatives of $E$ with respect to the weights are taken in order to determine
the gradient. For the output layer the partial derivatives are given by

$$\frac{\partial E}{\partial w_{h,ij}} = -2 \sum_{k=1}^{p} \delta_{h,j}^{k} \lambda_{h-1,i}^{k}$$

(4.18)

where $h$ is the number of layers, $\lambda_{h-1,i}^{k}$ is the output for the $k$th training pattern of the $i$th hidden layer neuron which feeds into the output layer, and

$$\delta_{h,j}^{k} = (d_{j}^{k} - y_{j}^{k}) \phi'(\sum_{q=1}^{l_{k}} w_{h,q,i} \lambda_{h-1,q}^{k})$$

(4.19)

where $\phi'$ means the first derivative of the neural activation function. For every other layer the partial derivatives are given by

$$\frac{\partial E}{\partial w_{r,ij}} = -2 \sum_{k=1}^{p} \delta_{r,j}^{k} \lambda_{r-1,i}^{k}$$

(4.20)

where $1 \leq r \leq h - 1$, $\lambda_{r-1,i}^{k}$ represents the outputs of the $(r-1)^{th}$ hidden layer, $\lambda_{r,i}^{k} = u_{i}^{k}$ (the network inputs), and

$$\delta_{r,j}^{k} = (\sum_{q=1}^{n} \delta_{r+1,q}^{k} w_{r+1,jq}) \phi'(\sum_{i=1}^{m} w_{r,ij} \lambda_{r-1,i}^{k})$$

(4.21)

The error, $e_{h,j}^{k}$, at the output layer is defined.

$$e_{h,j}^{k} = d_{j}^{k} - y_{j}^{k}$$

(4.22)

where the "h" refers to the output layer and $1 \leq j \leq n$, and $1 \leq k \leq p$. The error at the hidden layers may then be defined:

$$e_{r,j}^{k} = \hat{\lambda}_{r,j}^{k} - \lambda_{r,j}^{k}$$

(4.23)

where $1 \leq r \leq h - 1$ and $\hat{\lambda}_{r,j}^{k}$ refers to the desired value of a hidden layer neuron’s output.

The goal is then to estimate the values of $\hat{\lambda}_{r,j}^{k}$. Comparing equations (4.19) and (4.22) shows that

$$\delta_{h,j}^{k} = e_{h,j}^{k} \phi'(\sum_{q=1}^{l_{k}} w_{h,q,i} \lambda_{h-1,q}^{k})$$

(4.24)
Then by comparing equations (4.21) and (4.23) and using the analogy suggested by equation (4.24), we make the following approximation:

\[ \varepsilon^k_{rj} \approx \sum_{q=1}^{n} \delta^k_{r+1,q} u_{r+1,jq} \]

which allows the activations of the hidden layers to be estimated using

\[ \dot{\lambda}^k_{r,j} \approx \lambda^k_{r,j} + \sum_{q=1}^{n} \delta^k_{r+1,q} w_{jq} \]

(4.25)

The change of variables for the hidden layer can then be computed using

\[ \Phi(V_r) = \dot{H}_r \]

(4.26)

where \( V_r \in \mathbb{R}^{p \times l_r} \) is the new matrix which maps on to \( \dot{H}_r \in \mathbb{R}^{p \times l_r} \), and the \( k_j^{th} \) element of \( H_r \) is equal to \( \lambda^k_{r,j} \). Note that equation (4.25) implies that this method of approximating the hidden layer desired activations has the same restriction on activation derivatives that backpropagation does.

A formal procedure for performing quadratic optimization with desired hidden layer output approximations for a two layer network appears in Algorithm 1. In [22] it is reported that the algorithm usually converges with a fairly small error within one iteration. The paper also deals with concerns over the magnitude of the weights and elements of the \( V \) matrices. For example, by using equation (4.25) it would be possible to assign values of \( H_D \) which were not within the range of the neurons’ activation function (e.g., an element of \( H_D \) being assigned a value greater than one when the sigmoid activation function was being used). Methods for scaling and bounding the hidden layer estimates are given in [22] to alleviate this problem.

Sartori and Antsaklis recommend that the algorithm be used for training classification problems rather than function approximation, due to the inexact nature of the approximations discussed above.
Algorithm 1 (Multilayer Quadratic Optimization).
Given $D$, find $V_2$ such that $\Phi(V_2) = D$.
Initialize $W_1$ and $W_2$ with small random values.
while trace $((Y - D)^T(Y - D)) > \text{Max. Error.}$ do
  Find $V_1$ using equations (4.25) and (4.26).
  Find $W_1$ by solving $U^T U W_1 = U^T V_1$.
  Find $H = \Phi(UW_1)$.
  Find $W_2$ by solving $H^T HW_2 = HV_2$. 
CHAPTER 5

BLOCK CONJUGATE RESIDUAL

The conjugate residual (CR) algorithm [6] is a Krylov subspace iterative algorithm for solving linear systems

$$A x = b$$

where $A \in \mathbb{R}^{m \times m}$ is symmetric positive definite, $x, b \in \mathbb{R}^m$ and $x$ is the unknown. It does not solve the problem directly like Gaussian Elimination routines, but instead solves

$$\min_x (A x - b)^T (A x - b)$$

Algorithm 2 details the standard conjugate residual algorithm. Note that the iterative loop specifies $m$ steps. This is because by the $m^{th}$ step the Krylov subspace will be equal to the entire range of $A$, and an exact solution is guaranteed. In practice, finding an "exact" solution in $m$ steps is dependent on the conditioning of the $A$ matrix, due to numeric (as opposed to analytic) difficulties [13]. This is why in most implementations of conjugate residual, the residual square error $(r_k)^T r_k$ is calculated during the solution, and this value is used as a stopping criterion instead of performing all $m$ steps. Practice shows that an acceptable error can often be obtained in a lot fewer than $m$ steps, and the linear system of equations is solved within a desired accuracy in a very short time.

A comparison of equations (2.1) and (5.1) shows definite similarities between the
Algorithm 2 (Standard Conjugate Residual).

input $A \in \mathbb{R}^{m \times m}$ s.p.d., $b \in \mathbb{R}^m$.

pick $x^0 \in \mathbb{R}^m$

set $r^0 = b - Ax^0$ ; Initial error (residual)

set $q^0 = r^0$ ; Initial search direction

for $k = 1$ to $m$

$\alpha_k = \frac{(r_k)^T A q_k}{(q_k)^T A q_k}$

$x^{k+1} = x^k + \alpha_k q^k$

$r^{k+1} = r^k - \alpha_k A q^k$

select $q^{k+1}$ (next search direction):

$\beta^{k+1} = \frac{(r^{k+1})^T A q^k}{(q^{k+1})^T A q^k}$

$q^{k+1} = r^{k+1} - \beta^{k+1} q^k$

single layer neural network training problem and the conjugate residual solution to a linear system of equations. Both problems involve a least mean squares (LMS) procedure for obtaining their respective solutions. Unfortunately there are also some striking differences, in equation (2.1), $W$ and $D$ are matrices of unequal size, while $x$ and $b$ are vectors of equal size in equation (5.1). Equation (2.1) requires a trace while (5.1) does not. Worst of all, equation (2.1) involves a nonlinear function performed on $UW$ that does not appear in a system of linear equations.

Recall from chapter 4 that it is possible to perform a change of variables such that $\Phi(V) = D$ which effectively removes the nonlinearity from the problem. It was shown, in that situation, that if an exact solution does exist, it is given by $W = (U^T U)^{-1} U^T V$. But even if $(U^T U)^{-1}$ does not exist, Krylov subspace techniques may still be used to find a $W$ which minimizes equation (4.2), thus producing the LMS solution.

The transformation of the single layer neural network training problem to conjugate residual is nearly complete. $U^T U$ is square with dimension $m$, just like the matrix $A$ in equation (5.1), and $W$ and $U^T V$ are of equal dimension, just like $x$ and $b$. However $W$ and $U^T V$ are matrices and equation (4.2) involves a trace. These
problems can be overcome with a block form of conjugate residual which solves for entire matrices instead of individual vectors. The form of block conjugate residual is nearly identical to standard conjugate residual except that matrix traces are used to compute $\alpha$, $\beta$ and the residual error.

The method for solving equation (4.2) through block conjugate residual\(^1\) is given in Algorithm 3. As mentioned above, finding a $W$ that satisfies equation (4.2) is equivalent to finding a $W$ which solves $U^T U W = U^T V$, however, in Algorithm 3, the initial residual, $r^0$, and the initial search “direction”\(^2\), $q^0$, are defined according to

$$r^0 = V - U W$$

$$q^0 = U^T r^0$$

instead of

$$r^0 = U^T V - U^T U W$$

$$q^0 = r^0$$

This is because we want the residual square error used in the algorithm to be equal to the residual square error defined in equation (4.2). During the iterations of the algorithm, the current residual square error may be calculated according to

$$\text{square error} = \text{trace } ((r^k)^T r^k)$$

which can be used to form a stopping criterion instead of performing all $m$ iterations that appear in the algorithm.

### 5.1 Analysis of Error Convergence

The error defined by equation (5.4) is appropriate for use during training so that the error measure is equal to that defined by equation (4.2). However the Krylov

\(^1\)Section A.3 of the appendix contains the C function \texttt{conjres()} which implements algorithm 3. The function is part of the \texttt{makenet} program.

\(^2\)“Direction” appears in quotes because, in the block case, $q^k$ is a matrix not a vector.
Algorithm 3 (Block Conjugate Residual).

| input \( U \in \mathbb{R}^{p \times m}, V \in \mathbb{R}^{p \times n} \). |
| pick \( W^0 \in \mathbb{R}^{m \times n} \). |
| set \( r^0 = V - UW^0 \); Initial error (residual) |
| set \( q^0 = UT^0 \); Initial search "direction" |
| for \( k = 1 \) to \( m \) |
| \( \alpha^k = \frac{\text{trace} \left( (Uq^k)^T r^k \right)}{\text{trace} \left( (Uq^k)^T Uq^k \right)} \) |
| \( W^{k+1} = W^k + \alpha^k q^k \) |
| \( r^{k+1} = r^k - \alpha^k U q^k \) |
| select \( q^{k+1} \) (next search direction): |
| \( \beta^k = \frac{\text{trace} \left( UU^T r^{k+1} \right) Uq^k}{\text{trace} \left( (Uq^k)^T Uq^k \right)} \) |
| \( q^{k+1} = UT^0 r^{k+1} - \beta^{k+1} q^k \); orthogonal to all previous \( q^k \)'s |

Subspace searched by the algorithm is given by

\[ K^k(U^T U, U^T r^0) = \text{span} \{ U^T r^0, (U^T U)U^T r^0, \ldots, (U^T U)^k U^T r^0 \} \]

where \( r^0 \) is given by equation (5.2), and the superscript \( k \) refers to the \( k \)th iteration of the algorithm. An examination of the Krylov subspace shows that the algorithm is solving

\[ U^T UW = U^T V \quad (5.5) \]

for \( W \) and is analytically guaranteed a solution with 0 error in \( m \) steps, assuming that \( U^T U \) is full rank. This result is true because at the \( m \)th step the Krylov subspace \( K^m(U^T U, U^T r^0) \) will be equal to the entire range of \( U^T U \), as seen by a simple application of the well known Cayley-Hamilton theorem of linear systems [14].

As mentioned previously, solving (5.5) is equivalent to finding the LMS solution of \( UW = V \), which may have no exact solution, yielding a nonzero final error value for trace \( ((r^m)^T r^m) \). In order to analyze the error convergence of the algorithm, it is appropriate to base the error and the residual on the actual Krylov subspace. The residual is then defined as

\[ e^k = U^T V - U^T UW^k \]

\(^3\)Instead of \( r^k = V - UW^k \).
where $e^k \in \mathbb{R}^{m \times n}$. The associated square error is given by $\text{trace } ((e^k)^T e^k)$, and the Krylov subspace for the $k^{th}$ iteration can be rewritten as

$$
K^k(U^T U, e^0) = \text{span } \{e^0, (U^T U)e^0, \ldots, (U^T U)^k e^0\}
$$

The derivation of error convergence for block conjugate residual is very similar to that for standard conjugate residual [6], but since the algorithms are not exactly the same, a complete derivation will be given here.

The initial residual is given by

$$
e^0 = U^T V - U^T U W^0
$$

and the weight update rule at the $k^{th}$ iteration is given by

$$
W^{k+1} = W^0 + t^k (U^T U) e^0
$$

where $t^k \in \mathcal{P}^k$ is a $k^{th}$ degree polynomial resulting from the range of $K^k(U^T U, e^0)$, the Krylov subspace at the $k^{th}$ iteration. The residual at the $k^{th}$ iteration is then

$$
e^{k+1} = U^T V - U^T U W^{k+1}
$$

$$
= U^T V - U^T U (W^0 + t^k (U^T U) e^0)
$$

$$
= U^T V - U^T U W^0 - U^T U t^k (U^T U) e^0
$$

$$
= e^0 - U^T U t^k (U^T U) e^0
$$

$$
= (I - U^T U t^k (U^T U)) e^0
$$

which we can write as

$$
e^k = \Psi^k(U^T U)
$$

where $\Psi^k \in \mathcal{P}^k$ is a $k^{th}$ degree polynomial and $\Psi(0) = 1 \; \forall k$.

The square residual error is then given by

$$
\text{trace } ((e^k)^T e^k) = \min_{\Psi^k \in \mathcal{P}^k, \Psi(0) = 1} \text{trace } (\Psi^k(U^T U) e^0)^T \Psi^k(U^T U) e^0
$$
the trace of $x^T x$ can be expressed as the sum of the squares of every element of the matrix $x$. Thus the last equation can be rewritten in terms of the Euclidean or two norm:

$$\text{trace } ((e^k)^T e^k) = \sum_{j=1}^{n} \|e_j^k\|^2 = \min_{\Psi} \sum_{j=1}^{n} \|\Psi^k (U^T U) e_j^0\|^2$$

where $e_j^k$ refers to the $j^{th}$ column of $e$ at the $k^{th}$ iteration. The ratio between the error of the $k^{th}$ and $0^{th}$ iterations is now expressed as an inequality:

$$\sum_{j=1}^{n} \|e_j^k\|^2 \leq \min_{\Psi} \sum_{j=1}^{n} \|\Psi^k (U^T U)\|_2 \|e_j^0\|^2$$

$$\sum_{j=1}^{n} \|e_j^k\|^2 \leq \min_{\Psi} \|\Psi^k (U^T U)\|_2$$

where $\|T^k\|$ is an arbitrary $k^{th}$ degree polynomial where $T^k(0) = 1 \forall k$.

In order to find useful bounds for $\|T^k(U^T U)\|^2$ it is necessary to determine the effect of $U^T U$ on the norm and then to find an appropriate class of polynomials for $T^k$. First $U^T U$ will be examined. Assume that $U^T U$ is (symmetric) positive definite and diagonalizable. The matrix is obviously symmetric since it is computed from the transpose of $U$ multiplied by itself. The matrix must be full rank for it to be positive definite. This is not a restrictive assumption because $U^T U$ is dimension $m \times m$ while $U$ is $p \times m$ and the number of training patterns $p$ is almost always greater than the number of network inputs $m$. If $U^T U$ is diagonalizable then

$$U^T U = S \Lambda S^{-1}$$

where $S \in \mathbb{R}^{m \times m}$ is a matrix of the eigenvectors of $U^T U$ and $\Lambda \in \mathbb{R}^{m \times m}$ is a diagonal matrix with the corresponding eigenvalues along the diagonal. Since $U^T U$ is assumed to be positive definite, the columns of $S$ are orthogonal and thus may be constructed
as orthonormal. The individual eigenvalues $\lambda_i$ are greater than zero, where
\[ \Lambda = \begin{bmatrix} \lambda_1 & \cdots & \lambda_m \end{bmatrix} \]

Since $\Lambda$ is a diagonal matrix, its squared two norm is easily computed:
\[ \|\Lambda\|^2 = \max_{\|\mathbf{x}\|=1} \left( (\lambda_1 x_1)^2 + \cdots + (\lambda_m x_m)^2 \right) = \max_i \lambda_i^2 \]

where $i = 1, \ldots, m$. The squared two norm of $T^k(U^TU)$ is then
\[ \|T^k(U^TU)\|^2 = \|T^k(SAS^{-1})\|^2 = \|ST^k(\Lambda)S^{-1}\|^2 \leq \|S\|^2 \|T^k(\Lambda)\|^2 \|S^{-1}\|^2 \]

$T^k(\Lambda)$ is a diagonal matrix since $\Lambda$ is diagonal, thus $\|T^k(\Lambda)\|^2 = \max_i |T^k(\lambda_i)|^2$ where $i = 1, \ldots, m$. The columns of $S$ (eigenvectors of $U^TU$) are orthonormal, so $\|S\|_2 = \|S^{-1}\|_2 = 1$ and
\[ \frac{\text{trace} \left( (e^k)^T e^k \right)}{\text{trace} \left( (e^0)^T e^0 \right)} = \sum_{j=1}^{n} \frac{\|e_j^k\|^2}{\|e_j^0\|^2} \leq \max_i |T^k(\lambda_i)|^2 \tag{5.6} \]

We are now free to choose any $k^{th}$ degree polynomial for $T^k$ (as long as $T^k(0) = 1$) in order to develop a respectable bound for the error convergence. One good choice is the $k^{th}$ degree Chebyshev polynomials, $P_{\text{Cheb}}^k(x)$, which minimize the ripple of the polynomial within the range $x_{\text{min}}$ to $x_{\text{max}}$. These polynomials have the property that
\[ \max_{x \in [x_{\text{min}}, x_{\text{max}}]} |P_{\text{Cheb}}^k(x)| \leq 2 \left( \frac{x_{\text{max}} - x_{\text{min}}}{\sqrt{x_{\text{max}} - x_{\text{min}}} + 1} \right)^k \]

If we choose $T^k = P_{\text{Cheb}}^k$ and $x_{\text{min}} = \lambda_{\text{min}}, x_{\text{max}} = \lambda_{\text{max}}$ (the minimum and maximum eigenvalues of $U^TU$) then
\[ \max_{\lambda \in [\lambda_{\text{min}}, \lambda_{\text{max}}]} |T^k(\lambda)|^2 \leq 4 \left( \frac{\sqrt{\lambda_{\text{max}} - \lambda_{\text{min}}}}{\sqrt{\lambda_{\text{max}} - \lambda_{\text{min}}} + 1} \right)^{2k} \]
Note that the condition number $K(U^T U)$ of the symmetric positive definite matrix $U^T U$ is given by

$$K(U^T U) = \frac{\lambda_{\text{max}}}{\lambda_{\text{min}}}$$

thus the error convergence can be written in terms of the condition number:

$$\frac{\text{trace } (e^k)^T e^k}{\text{trace } (e^0)^T e^0} = \sum_{j=1}^{n} \frac{\|e_j^k\|^2}{\|e_j^0\|^2} \leq 4 \left( \frac{\sqrt{K(U^T U)} - 1}{\sqrt{K(U^T U)} + 1} \right)^{2k} \tag{5.7}$$

A different choice of the polynomial $T^k$ shows that the error will go to zero in $s$ steps if $U^T U$ has $s$ distinct eigenvalues. At the $s^{\text{th}}$ step, choose $T^s$ to be a polynomial with roots at all of the distinct eigenvalue locations, it is then obvious that the right hand side of equation (5.6) will go to zero. We can also see this by examining equation (5.7). If at every step we only use a first order Chebyshev polynomial and use the remaining root to zero out the greatest remaining eigenvalue, then by the $s^{\text{th}}$ step the maximum remaining eigenvalue will be equal to the minimum remaining eigenvalue, and the right hand side of equation (5.7) will go to zero.
CHAPTER 6

DEPENDENCE IDENTIFICATION

In recent years, research has gone into finding new methods for training multilayer neural networks which are totally distinct from backpropagation. In [23], it is shown that by choosing the number of hidden layer neurons equal to the number of patterns \( l = p \) in a two layer network, the input layer weights can be chosen at random. Recall that for a two layer network \( \Phi(UW_1) = H \) and \( U \in \mathbb{R}^{p \times m}, W_1 \in \mathbb{R}^{m \times l}, H \in \mathbb{R}^{p \times l} \). If \( p = l \), then \( H \) will be a square matrix. It is shown in [23] that if \( G \) is filled with random elements then \( H \) will almost always be full rank. The output layer requires \( \Phi(HW_2) = D \) or \( HW_2 = V \), and if \( H \) is square and full rank then an exact solution exists for the problem. This method is fast, but it requires an excessively high number of hidden layer neurons, and it eliminates the possibility for the first layer weights to directly help in solving the problem.

The purpose of the first layer of a neural network is to divide the pattern space with separating hyperplanes. A method for finding the equations of the separating hyperplanes and transforming these equations into first layer weights is given in [11]. This method would not require continuously differentiable activation functions and would determine the number of hidden layer neurons, but it is computationally expensive. The method may be used for classification and pattern recognition applications; it is not meant for function approximation.
In this chapter, a novel method called *dependence identification* (DI) for constructing and training multilayer feedforward neural networks is presented. The method does not require that the activation functions be continuously differentiable, and it determines the number of hidden layer neurons as a part of its output. The method works for both classification and function approximation, and it is much faster than backpropagation (as shown by the examples in chapter 7). An example is used to illustrate the method in section 6.1 before formally presenting the algorithm in section 6.2. Sections 6.3 and 6.4 deal with some problems with and enhancements to the dependence identification algorithm.

### 6.1 A Network Construction Example

A neural network is to be constructed which realizes the *exclusive or* (XOR) boolean function. The XOR function has two boolean inputs and a single boolean output. The desired input/output mapping is given below in table 6.1.

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.1: The XOR function

The input patterns are augmented with a vector of constant inputs to act as a bias. The $U$ and $D$ matrices are then

$$
U = \begin{bmatrix}
0 & 0 & 1 \\
0 & 1 & 1 \\
1 & 0 & 1 \\
1 & 1 & 1 \\
\end{bmatrix} \quad D = \begin{bmatrix}
0 \\
1 \\
1 \\
0 \\
\end{bmatrix}
$$
Because the I/O space is boolean, it makes sense to define the neural activation function $\phi$ as the step function which also has output values of 0 and 1. We now need to define the $V$ matrix such that $\Phi(V) = D$. As mentioned in chapter 4, the step function does not have a one-to-one inverse, but we can still assign values to $V$ which would give rise to the appropriate values in $D$. Whenever $D$ has a 0, a -1 is assigned to $V$, and whenever $D$ has a 1, a 1 is assigned to $V$. Thus

$$V = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

If a single layer neural network with weight matrix $W \in \mathbb{R}^{3 \times 1}$ is to realize the function, then we need $UW = V$. A solution to this equation exists iff

$$\text{rank } (U) = \text{rank } ([U \ V])$$

but $U$ is rank 3 and $[U \ V]$ is rank 4, thus a single layer neural network solution does not exist. The pattern space is not linearly separable. This is a well known result.

The fact that a single layer neural network can not realize the XOR function brought much criticism to the field of neural networks in its early years. If the XOR function is to be realized, the network must have a minimum of two layers.\(^1\)

The rank condition (6.1) dictates the entire pattern set can not be realized at once, but what about a subset? Take the first three rows of $U$, call them $U_{1:3}$, and the first three rows of $V$, call them $V_{1:3}$. Now $\text{rank } (U_{1:3}) = \text{rank } ([U_{1:3} \ V_{1:3}]) = 3$ so a solution to this problem does exist. An inspection of the three patterns shows that the solution is simply the boolean OR function. The equation

$$U_{1:3}g_1 = V_{1:3}$$

\(^1\)If only non-decreasing activations functions are allowed then a single layer neural network can not realize the XOR function. Single layer solutions to the problem do exist if functions like the Gaussian or $\phi(x) = \text{step}(x + \alpha) - \text{step}(x - \alpha)$ are permitted.
Figure 6.1: Components of the XOR neural network.

is solved for $g_1$ and we obtain

$$g_1 = \begin{bmatrix} 2 \\ 2 \\ -1 \end{bmatrix}$$

Weight matrix $g_1$ provides a single layer neural network solution to the boolean OR problem as shown in Figure 6.1a.

Now we take the last three rows of $U$ and $V$: $U_{2:4}, V_{2:4}$. Once again the two matrices satisfy the rank condition of equation (6.1) and we can see that the solution would be the boolean NAND function. The equation

$$U_{2:4}g_2 = V_{2:4}$$

is solved for $g_2$:

$$g_2 = \begin{bmatrix} -2 \\ -2 \\ 3 \end{bmatrix}$$

The single layer solution to the boolean NAND problem is shown in Figure 6.1b.

The two single layer neural networks for OR and NAND can be used as building
blocks for the multilayer XOR network. First we will define another single layer network which has an output that never changes. This will act as a bias just like the third column of \( U \). The weight matrix for this bias neuron is then

\[
g_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}
\]

The first layer weight matrix can then be constructed as

\[
W_1 = [g_1 \ g_2 \ g_3]
\]

The output of the hidden layers is then computed:

\[
\Phi(UW_1) = H = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix}
\]

and it is easy to see that \( \text{rank}(H) = \text{rank}([H \ V]) = 3 \), thus a solution \( W_2 \in \mathbb{R}^{3 \times 1} \) to the final problem \( HW_2 = V \) does exist.

\[
W_2 = \begin{bmatrix} 2 \\ 2 \\ -3 \end{bmatrix}
\]

The network which realizes the XOR problem is shown in Figure 6.1c.

6.2 The Dependence Identification Algorithm

The XOR network in the previous section was constructed with the dependence identification algorithm. The algorithm constructs a network as follows: First attempt to create a single layer neural network by solving problem (4.2) using the method of quadratic optimization. Compute the error of the single layer network.
If the error is acceptable, then training is done. Otherwise create a layer of hidden neurons which get portions of the output matched correctly, i.e., every pattern should be classified correctly by at least one hidden layer neuron. To do this choose an \( m \times m \) portion (it is assumed that there are more training patterns than there are inputs, i.e., \( p > m \)) of \( U \), and the corresponding \( m \times n \) portion of \( D \). Use these patterns to solve a single layer neural network training problem as above. It is guaranteed that a zero error solution to this problem exists if the \( m \times m \) portion of \( U \) is full rank. Next test the network on all the remaining patterns and keep track of those patterns which are not mapped correctly. Choose \( m \) patterns which failed the test and repeat the process until every pattern is matched correctly by at least one hidden layer neuron. The input patterns \( U \) are then redefined to be the output of the hidden layer neurons. Repeat the procedure, with the new set of input patterns, until a maximum number of layers has been reached or the network error is within the desired tolerance.

The method is presented formally in Algorithm 4. The inputs to the algorithm include the training patterns \( U \) and \( D \) as well as the neural activation function \( \phi \), the maximum number of layers \( h_{\text{max}} \), and two tolerances \( \epsilon_N \) and \( \epsilon_p \). The acceptable error for the entire network is given by \( \epsilon_N \). The other tolerance, \( \epsilon_p \), is used to determine whether the output of the network for an individual pattern is within bounds. Larger values for \( \epsilon_p \) give fewer hidden layer neurons. The systems of linear equations are efficiently solved using block conjugate residual (see chapter 5).

The dependence identification algorithm constructs a network which solves the single layer problem within the desired accuracy, if such a solution actually exists. The number of layers created can be bounded with the parameter \( h_{\text{max}} \). The number of hidden layer neurons depends on the number of layers and the desired accuracy. It is known that \( p \) neurons in the hidden layer suffice to store \( p \) arbitrary patterns in
a two layer network; see [2], [19], and [23]. Assuming the $m \times m$ portions of $U$ are full rank the dependence identification algorithm has an upper bound of

$$l_{\text{max}} = \text{ceiling}(\frac{p}{m})n + 1$$

(6.2)

where "ceiling (x)" means the smallest integer greater than or equal to $x$. The dependence identification algorithm assumes that one input is a constant used to form thresholds for the first layer's neurons. Thus for a SISO system $m = 2$ and $n = 1$ and the bound on the number of hidden layer neurons created by dependence identification is less than the bound of $p$.

The algorithm's solution is based on solving a succession of systems of linear equations. Krylov subspace methods, like block conjugate residual, are recommended since they are iterative methods for solving linear equations, working toward minimizing a quadratic error function instead of attempting to solve the problem exactly in a single complicated step. These methods are guaranteed to converge in a number of steps equal to the order of the system, assuming that the problem is sufficiently well conditioned [13]. The speed of these algorithms is a key to the overall speed of dependence identification.

The dependence identification algorithm can not only be used to construct continuous sigmoidal networks but can also be used to construct networks that use discontinuous switching functions. The discontinuities of these functions prevent them from being used with backpropagation training.

The dependence identification algorithm is presented as a serial method for determining a network architecture and weight values. Algorithms like backpropagation are inherently parallel and may be realized in parallel by hardware. However backpropagation has no built-in method for determining an appropriate network architecture or initial conditions. Dependence identification can be used to determine

---

2The bound of $p$ assumes that one input to the network is constant. If the neural thresholds are handled separately, then the bound is $p - 1$. 

48
these quantities, which would then be used to construct an actual parallel BP net. Section 7.3 gives an example of this and the overall usefulness of the algorithm for presenting backpropagation with an appropriate set of initial conditions.

6.3 Increased Tolerance and Recomputed Weights

There may be situations where dependence identification determines a number of neurons and/or layers that may not be physically represented due to hardware or software constraints. The number of layers may be limited by the parameter $h_{\text{max}}$, but it is not so easy to limit the number of neurons within a hidden layer. One way to decrease the number of hidden layer neurons is to increase the tolerance $\epsilon_p$. This value is used to determine whether a pattern should be considered part of a linearly dependent set, i.e., if a pattern subset $u, d, v$ and associate weight $g$ satisfies $ug = v$, then another pattern $\hat{u}, \hat{d}, \hat{v}$ is considered "linearly dependent" if the elements of $|\Phi(\hat{u}g) - \hat{d}|$ are all less than $\epsilon_p$. Linear dependence is written in quotes here because the nonlinear, not the linear, function is actually being used to determine a pattern match. If the linear function$^3$ were used, then the check would be to see if the elements of $|\hat{u}g - \hat{v}|$ are all less than $\epsilon_p$. Increasing the value of $\epsilon_p$ increases the number of patterns that are considered linearly dependent no matter which method is used to check for linear dependence. Since each group of $n$ hidden layer units is associated with a single linearly dependent pattern set, increasing $\epsilon_p$ decreases the number of hidden layer neurons.

Unfortunately when the number of hidden layer neurons are decreased by including more patterns in the same group, the overall error of the network tends to increase. One way to reduce this error is to recompute the weight matrix $g$ after the complete set of linearly dependent patterns has been found. The original value

$^3$The makenet program presented in the appendix allows either the linear or the nonlinear function to be used.
of \( g \) is computed just as before, using an \( m \times m \) section of \( U \). The set of linearly dependent patterns is also computed as before, but then \( g \) is recomputed using the complete set of linearly dependent patterns. This is easily done with the block conjugate residual algorithm since it does not require that the \( u \) matrix be square. An example of decreasing the number of hidden layer neurons while keeping the error low with recomputed weights is given in section 7.2.

### 6.4 Discussion of Linear System Solutions

Most of the material in the previous sections assumes that the individual \( m \times m \) portions of \( U \) will be full rank and thus lead to a unique solution for each \( g \). If the portions are not full rank, then there are either an infinite number or zero exact solutions for \( g \). There is no problem in the case where an infinite number of solutions for \( g \) exist, but problems may occur when there are no solutions. The block conjugate residual (BCR) algorithm will return an answer for \( g \) that minimizes trace \((ug - v)^T(ug - v)\), but this value will be nonzero if an exact solution for \( g \) does not exist. BCR is an iterative technique and requires its own tolerance for determining how accurate of a solution it should generate. If this tolerance is too high, or it is relatively high compared to the tolerance \( \epsilon_p \), then it is possible that the solution \( g \) obtained by BCR will not be close enough to match any patterns even if the \( m \times m \) portion of \( U \) is full rank. If no exact solution for \( g \) exists or a poor solution is found due to a bad tolerance for BCR, then it is possible that the number of patterns which make up a linearly dependent set will be zero and the algorithm has "balked." If the algorithm as presented in section 6.2 does not check for this situation, then it is possible that it will enter an infinite loop, constantly trying to solve the same unsolvable system of linear equations over and over again.

In order to avoid this problem, determine if the solution \( g \) was not able to match

---

4The term "balk" is used in the makenet program given in the appendix.
any patterns correctly. If it was not, then try computing $g$ with a portion of $U$ that is not square, e.g., choose an $(m-1) \times m$ portion of $U$ instead of an $m \times m$ portion. If still none of the patterns are correctly matched, then choose $m-2$ patterns, $m-3$ patterns, etc. . . . At some point the algorithm will finally find a weight matrix $g$ that matches at least one of the unmatched patterns correctly, at which point the program can continue normally with the remaining patterns. If the tolerance for BCR is too high, then it may not be able to find an appropriate value for $g$ for even one pattern. In this case the tolerance for BCR should be lowered.

This discussion points out a major concern with the dependence identification algorithm: determining which $m \times m$ portions of $U$ to use as the initial seeds for creating the $g$ matrices. Even if all of the $m \times m$ portions of $U$ are full rank, it is still possible that a different choice would lead to better results. The makenet program given in the appendix simply chooses the groups based on the order that the program reads them from the input file. For function approximation, it may be a good idea to sort the patterns according to the norm of the individual input patterns. This would cause patterns that are relatively "close" together to be part of the same $u$ matrix. This method would probably not be appropriate or useful for classification problems (like the parity problem discussed in section 7.3). Exploration of the choice of useful pattern subsets is left as a future research topic.
Algorithm 4 (Dependence Identification).

input $U \in \mathbb{R}^{p \times m}$, $D \in \mathbb{R}^{p \times n}$, $\phi: \mathbb{R} \rightarrow \mathbb{R}$, $\epsilon_N$, $\epsilon_p$, $h_{\text{max}}$.

Create $V$ such that $\Phi(V) = D$

Set $h = 1$

repeat

Solve $\min_{W_h} \text{trace} \left((UW_h - V)^T (UW_h - V)\right)$, $W_h \in \mathbb{R}^{m \times n}$.

Compute error $e = \text{trace} \left((\Phi(UW_h) - D)^T (\Phi(UW_h) - D)\right)$

if ($e > \epsilon_N$) and ($h < h_{\text{max}}$) then

Set $G = \text{empty matrix}$, $l = 0$

Set $U_{\text{unmarked}} = U$, $V_{\text{unmarked}} = V$

repeat

Choose $u \subset U_{\text{unmarked}}$, $v \subset V_{\text{unmarked}}$ where $u \in \mathbb{R}^{m \times m}$, $v \in \mathbb{R}^{m \times n}$

if less than $m$ patterns are unmarked then

augment $u$ and $v$ with previously marked patterns.

Solve $\min_{g} \text{trace} \left((ug - v)^T (ug - v)\right)$, $g \in \mathbb{R}^{m \times n}$.

Add the solution $g$ as new column(s) of $G$.

$l \leftarrow l + n$

Mark all patterns which satisfy $U_{\text{unmarked}} g = V_{\text{unmarked}}$ within the tolerance $\epsilon_p$.

until all patterns have been marked.

Find the hidden layer outputs $H = \Phi(UG) \in \mathbb{R}^{p \times l}$.

Add a zero column to $G$ with a single nonzero element corresponding to the constant input.

$W_h \leftarrow G$

$U \leftarrow H$

$m \leftarrow l$

$h \leftarrow h + 1$

until ($e \leq \epsilon_N$) or ($h = h_{\text{max}}$)

output $W_i$ for $i = 1, \ldots, h$ and the error $e$
CHAPTER 7

EXAMPLES AND SIMULATIONS

Programs are written in C to perform backpropagation (BP) and dependence identification\(^1\) (DI) on two layer feedforward neural networks. The programs are run on Sun SPARCstation 2 workstations. Section 7.1 contains comparisons between networks constructed and trained with dependence identification to those trained with backpropagation. The systems trained include the XOR function (section 7.1.1), a sine wave (section 7.1.2), a 2 input/2 output continuous function (section 7.1.3), and a 3 input/1 output continuous function (section 7.1.4). Section 7.2 shows how the number of hidden layer neurons may be reduced by increasing the DI tolerance value \(\epsilon_p\). The overall error of the network is kept down, while the tolerance is increased, by recomputing the weights after each linearly dependent set of training patterns is found. Section 7.3 shows how dependence identification may be successfully used to find a network architecture and set of initial conditions for further training by backpropagation.

\(^1\)The source code for the program which performs dependence identification (makenet.c) is located in the appendix.
7.1 Dependence Identification vs. Backpropagation.

7.1.1 The XOR Problem

The derivation of the XOR network using dependence identification was given in section 6.1. When the network was constructed on the computer it achieved a final error of zero, computing the solution in less than 0.01 seconds.\(^2\) Equation (6.2) gives \(l_{\text{max}} = 3\), and all three hidden layer neurons were required for this example.

The problem was also solved using backpropagation (BP). Recall that backpropagation requires continuously differentiable activation functions, thus the step function that was used by dependence identification cannot be used by BP. Instead, \(\phi(x) = \tanh(x)\), the hyperbolic tangent function is used. The learning rates chosen were \(\alpha = 0.2, \beta = 0.2\). Standard BP must be told the number of hidden layer neurons in the network, so three are chosen: the same number that dependence identification "deduced". Thus the network architecture is 3 – 3 – 1, meaning 3 inputs (1 constant), 3 hidden layer neurons, and 1 output. The training is stopped when each of the network outputs is within 0.1 of the corresponding desired outputs. The final square error for BP is 0.0193, and the solution was calculated in 0.1667 seconds. The solution results are shown in table 7.1.

<table>
<thead>
<tr>
<th>(u_1)</th>
<th>(u_2)</th>
<th>Desired Output</th>
<th>DI Network Output</th>
<th>BP Network Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0403</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.9034</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0.9199</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0.0876</td>
</tr>
</tbody>
</table>

Table 7.1: Comparative results for the XOR problem.

\(^2\)The timing of the makenet program is only accurate to the nearest one hundredth of a second. The program actually reported that the problem was solved in 0.00 seconds.
7.1.2 Sine Wave Approximation

In the next example, the two training methods are used to approximate a sine wave, i.e.,

\[ d = \sin(u_1) \]
\[ u_1 \in [0, 4\pi] \]

Twenty training patterns are generated, evenly spaced from \( u_1 = 0 \) to \( u_1 = 4\pi \). The activation function for both networks is \( \phi(x) = \tanh(x) \).

Dependence identification reached a solution fairly quickly and produced a final square error of 0.2923 in 0.0167 seconds, while deriving a network architecture of 2 - 9 - 1. Equation (6.2) gives \( l_{\text{max}} = 11 \), so the actual value of \( l \) was less than \( l_{\text{max}} \).

Backpropagation had significant difficulties with learning the sine wave, being particularly susceptible to local minima in the error space. Many methods were tried to overcome the problem, including varying the number of hidden layer neurons. The problem was finally overcome by using a network architecture of 2 - 10 - 1 and setting \( \alpha = 0.02, \beta = 0.1 \). These learning parameters caused the solution to vary wildly, but it was stopped as soon as it found one set of weights that “beat” the local minimum. These weights were then used as the initial conditions with a slower learning rate of \( \alpha = 0.002, \beta = 0.1 \) and BP iterations were performed until the final square error was less than 1.0. The results are summarized in table 7.2. Note that dependence identification was over 1600 times faster than backpropagation and found a better solution!

The solutions were tested by choosing 126 values of \( u_1 \) evenly spaced between 0 and \( 4\pi \). The results are shown in figure 7.1. The twenty training patterns are marked

\footnote{The process of determining \( \alpha, \beta \), the number of hidden layer neurons and the stopping requirement for backpropagation is an exhausting trial and error procedure. The time required to actually determine an appropriate set of parameters is not included in the reported times for solutions using backpropagation.}
<table>
<thead>
<tr>
<th>Network Architecture</th>
<th>Square Error (Training)</th>
<th>Square Error (Test Set)</th>
<th>Time to Solution (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI 2 – 9 – 1</td>
<td>0.2923</td>
<td>1.8023</td>
<td>0.0167</td>
</tr>
<tr>
<td>BP 2 – 10 – 1</td>
<td>0.9993</td>
<td>3.8570</td>
<td>274.8</td>
</tr>
</tbody>
</table>

Table 7.2: Comparative results for the sine wave problem.

with ‘o’’s on the graph, and the corresponding learned responses to these patterns are marked with ‘x’’s. Table 7.2 shows that DI did better at matching all of the 126 test patterns as well as performing better on the training set of twenty patterns. This gives evidence that dependence identification can be a valid training method for approximating continuous functions with continuous neural activation functions, as opposed to the boolean function and step activations that were dealt with in sections 6.1 and 7.1.1.

Figure 7.1: Testing results of sine wave approximation.
7.1.3 2 Input/2 Output Continuous Function

In this example, the training methods are used to approximate the following 2 input/2 output sinusoidal function:

\[
\begin{align*}
    d_1 &= \frac{1}{5}(5\sin(u_1) + \cos(u_2)) \\
    d_2 &= \frac{1}{5}(3\cos(u_1) + 2\sin(u_2)) \\
    u_1, u_2 &\in [0, 2\pi]
\end{align*}
\]

The activation function for both networks is \( \phi(x) = \tanh(x) \). A training set is created by generating 200 uniformly distributed random values of \( u_1 \) and \( u_2 \) and calculating the associated values of \( d_1 \) and \( d_2 \). The learning parameters for backpropagation are \( \alpha = 0.005, \beta = 0.1 \), and the BP iterations are halted when the overall square error is less than 0.8. The results are summarized in table 7.3. Note that \textit{dependence identification derived a solution almost 300 times faster than backpropagation.} Equation (6.2) gives \( l_{\text{max}} = 135 \), while the actual value of \( l \) is 103: more than 20% less than \( l_{\text{max}} \).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
Network Architecture & Square Error (Training) & Square Error (Test Set) & Time to Solution (seconds) \\
\hline
DI & 3 - 103 - 2 & 0.3547 & 1.5508 & 15.14 \\
BP & 3 - 103 - 2 & 0.7902 & 2.1196 & 4429.25 \\
\hline
\end{tabular}
\caption{Comparative results for the 2 input/2 output sinusoidal function.}
\end{table}

Figure 7.2 shows the results of testing the system with \( u_1 \) and \( u_2 \) set to four different parametrized functions of \( t \). In all cases, \( t \) takes on 100 evenly spaced values from 0 to 2\( \pi \). The values of the inputs for the four different graphs in figure 7.2 are then:

for figure 7.2a:

\[ u_1(t) = t \]
\[ u_2(t) = \begin{cases} 
2t & 0 \leq t < \pi \\
4\pi - 2t & \pi \leq t \leq 2\pi 
\end{cases} \]

for figure 7.2b:

\[ u_1(t) = t \]

\[ u_2(t) = \pi (\sin(t) + 1) \]

for figure 7.2c:

\[ u_1(t) = t \]

\[ u_2(t) = 5\text{step}(t) - 4\text{step}(t - 2) + 4\text{step}(t - 4) - \ldots \]

(An oscillating “clock” function between 1 and 5 with a period of 2.)

for figure 7.2d:

\[ u_1(t_i) = u_1(t_{i-1}) + \text{random} \]

\[ u_2(t_i) = u_2(t_{i-1}) + \text{random} \]

There are a total of 400 points on which the solutions are tested (4 different input functions with 100 points on each). Table 7.3 shows that dependence identification performs better on the test sets as well as the training set of 200 points.

### 7.1.4 3 Input/1 Output Continuous Function

In this example, the training methods are used to approximate the following 3 Input/1 Output function:

\[ d = \frac{1}{10}(e^{u_1} + u_2 u_3 \cos(u_1 u_2) + u_1 u_3) \]

\[ u_1 \in [0,1] \]

\[ u_2, u_3 \in [-2,2] \]
Figure 7.2: Testing results of 2 input/2 output sinusoidal function.

Note that the function is highly nonlinear, involving an exponential, three multiplications and a trigonometric function.

The activation function for both networks is $\phi(x) = \tanh(x)$. A training set is created by generating 2000 uniformly distributed random values of $u_1, u_2$ and $u_3$ and calculating the associated values of $d$. The learning parameters for backpropagation are $\alpha = 0.005, \beta = 0.1$, and the solution is halted after one million BP iterations. The results are summarized in table 7.4. Note that the number of hidden layer neurons, $l = 286$, is more than 40% less than the bound of 501 given by equation (6.2).

In order to help backpropagation reach a solution with acceptable error in a minimal amount of time, BP only deals with 50 hidden layer neurons as opposed to the 286 hidden layer neurons derived by DI. Note that DI also has to perform
conjugate residual on matrices with rather large sizes, e.g., the $H$ matrix is $2000 \times 286$. But still DI is able to come up with a solution that has less error requiring only about one tenth of the time used by BP.

<table>
<thead>
<tr>
<th>Network Architecture</th>
<th>Square Error (Training)</th>
<th>Square Error (Test Set)</th>
<th>Time to Solution (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>0.4495</td>
<td>0.1221</td>
<td>291.13</td>
</tr>
<tr>
<td>BP</td>
<td>1.2661</td>
<td>0.2475</td>
<td>3026.93</td>
</tr>
</tbody>
</table>

Table 7.4: Comparative results for the 3 input/1 output system.

Figure 7.3 shows the results of testing the system with $u_1, u_2$ and $u_3$ set to four different parametrized functions of $t$. In all cases $t$ takes on 100 evenly spaced values from 0 to 1. The values of the inputs for the four different graphs in figure 7.3 are then:

for figure 7.3a:

$$u_1(t) = t$$
$$u_2(t) = 1.61 \quad \text{(a constant)}$$
$$u_3(t) = \begin{cases} 
8t - 2 & 0 \leq t < \frac{1}{2} \\
-8t + 6 & \frac{1}{2} \leq t \leq 1 
\end{cases}$$

for figure 7.3b:

$$u_1(t) = t$$
$$u_2(t) = \begin{cases} 
8t - 2 & 0 \leq t < \frac{1}{2} \\
-8t + 6 & \frac{1}{2} \leq t \leq 1 
\end{cases}$$
$$u_3(t) = \text{step}(t) - 2\text{step}(t - 0.25) + 2\text{step}(t - 0.5) - \ldots$$

(An oscillating "clock" function between 1 and -1 with a period of 0.25.)

for figure 7.3c:

$$u_1(t) = t$$
\[
\begin{align*}
  u_2(t) & = \text{step}(t) - 2\text{step}(t - 0.25) + 2\text{step}(t - 0.5) - \ldots \\
  u_3(t) & = 2\sin(4\pi t)
\end{align*}
\]

for figure 7.3d:

\[
\begin{align*}
  u_1(t) & = t \\
  u_2(t_i) & = u_2(t_{i-1}) + \text{random} \\
  u_3(t_i) & = u_3(t_{i-1}) + \text{random}
\end{align*}
\]

There are a total of 400 points on which the solutions are tested (4 different input functions with 100 points on each). Table 7.4 shows that dependence identification performs better on the test sets as well as the training set of 2000 points.

Figure 7.3: Testing results of 3 input/1 output system.
7.2 Increased Tolerance and Recomputed Weights

Hardware and/or memory restrictions may limit the number of hidden layers or hidden layer neurons that may be used in a particular neural network. Section 6.3 mentions that it is possible to decrease the number of hidden neurons created by the dependence identification algorithm by increasing the tolerance, $\varepsilon_p$, used to check whether or not a pattern should be included in a set of "linearly dependent" patterns. Unfortunately as the parameter $\varepsilon_p$ is increased and the number of hidden layer neurons decreases, the overall error of the network tends to increase. The increase in network error can be combated by recomputing the weight matrix $g$ (see algorithm 4) for the entire set of linearly dependent patterns after this set has been found. Figure 7.4 shows how the number of hidden layer neurons decreases as the tolerance $\varepsilon_p$ is increased from 0.05 to 5.00. The training patterns are for the sine wave approximation problem from section 7.1.2 and the network is constrained to have three layers ($h_{\text{max}} = 3$). Note that in this example the dependence identification algorithm determines linear dependence on the basis of the quadratic error cost, i.e., it is checking whether or not $ug = v$ instead of $\Phi(ug) = d$. This allows the error tolerance to be increased past $\varepsilon_p = 2$ up to $\varepsilon_p = 5$. In section 7.1.2 the dependence judgements were based on the nonlinear, not the quadratic check. Figure 7.4 shows that the number of hidden layer neurons can be successfully decreased with and without the recomputation of the weights. The reason the number of hidden layer neurons is different when the weights are recomputed is that a three layer network is being constructed. The inputs produce a set number of neurons in the first hidden layer dependent on $\varepsilon_p$ only, not on whether the weights are recomputed. The recomputation of weights produces different hidden layer activations (outputs) than those produced when the weights are not recomputed. Thus the second hidden

\footnote{The networks in section 7.1.2 have two layers}
layer may have a different number of neurons when the weights are recomputed since the inputs to the second hidden layer are different.

Figure 7.4: Decrease of hidden layer neurons due to increased tolerance $\varepsilon_p$.

Figure 7.5 shows how the overall error of the network varies as the tolerance is increased. The figure shows how recomputing the weight matrix helps to keep the overall error of the network from blowing up. It also shows that there is no simple relationship between the overall error and the dependence tolerance $\varepsilon_p$. For example, when $\varepsilon_p$ increases from 3 to 4, the overall error when the weights are not recomputed actually decreases from about 9.2 to 3.2. However the overall trend is that the recomputation keeps the error down. The highest error when the weights are recomputed is 2.35 (for the same value of $\varepsilon_p$ that causes an error of 9.25 when the weights are not recomputed) and is the only point at which the error exceeds 2. The average error for the fourteen different tolerances used in the figure is 3.47 when the weights are not recomputed and 1.25 when they are. It is possible that an error of 1.25 may be higher than the degree of approximation accuracy required for a particular problem. The next section shows how dependence identification can
be used to identify an appropriate network architecture and set of initial conditions that will allow backpropagation to quickly decrease the overall network error.

![Graph showing overall network error vs. increased tolerance εₚ.](image)

**Figure 7.5:** Overall network error vs. increased tolerance $\varepsilon_p$.

### 7.3 Dependence Identification for Backpropagation Initial Conditions

Backpropagation is extremely sensitive to initial conditions and there are some problems that are inherently very difficult for backpropagation to solve due to a large number of local minima and relatively flat sections of the cost function. One of these difficult problems is the parity checker [25]. A parity checker receives a number of boolean inputs. In this example, the boolean values are 1 and -1. The parity checker has one boolean output which should be 1 when there is an odd number of 1’s at the input and -1 when there are an even number of 1’s at the input.

A six input parity checker is used in this problem. The number of network inputs is seven because one input is constant and is used to create biases. The number of training patterns is 64 ($2^6 = 64$) which includes all possible combinations of six
boolean inputs. Dependence identification is used to find an appropriate two layer network architecture of $7 - 7 - 1$. Networks are then trained using backpropagation with random initial conditions and with the initial conditions obtained from dependence identification. The attempt to train the network with random initial conditions was tried multiple times with similar results at every attempt. Table 7.5 shows the results. Patterns are considered to be identified correctly if the network has the proper sign, i.e., a desired output of 1 is classified correctly if the network output is greater than zero and a desired output of $-1$ is classified correctly if the network output is less than zero. The entry for “% Error” is calculated as the number of missed patterns divided by 64.

The table shows that, by itself, dependence identification does not produce a very successful solution to the problem. However the initial conditions it produces are quite good for improving the network with backpropagation, bringing the error down from about 27% to just over 3%. When random initial conditions are used, backpropagation is completely unsuccessful in learning the problem, misclassifying more than half of the input space.

<table>
<thead>
<tr>
<th>Training Method</th>
<th>Correctly Classified Patterns</th>
<th>% Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP with initial conditions from DI</td>
<td>62</td>
<td>3.12%</td>
</tr>
<tr>
<td>Dependence Identification</td>
<td>47</td>
<td>26.56%</td>
</tr>
<tr>
<td>BP with random initial conditions</td>
<td>30</td>
<td>53.12%</td>
</tr>
</tbody>
</table>

Table 7.5: Results of parity checker training

Figure 7.6 shows how the error decreases with the iterations of backpropagation. When initial conditions from dependence identification are used, the error decreases from about 45 to level off around 8. Random initial conditions start out with a high error, and each iteration of backpropagation causes an extremely small change in the overall error. The error curve for random initial conditions (ICs) looks like a straight
line with error 64 on the graph, but a magnification would show that the error is slowly going up and down by very slight amounts. The random initial conditions occur in an area of the cost function with a very small slope and probably near a local minimum since the curve goes up and down, instead of steadily down. The graph shows 200,000 iterations (which required approximately 100 seconds to calculate). The actual training was continued for one million iterations with no change in the behavior of the curve with random initial conditions.

Figure 7.6: Network error vs. BP iterations for the parity checker problem.
A new method for training feedforward multilayer neural networks has been presented, and examples show that it works well for creating neural network approximations to continuous, as well as boolean, functions. The new method is faster than backpropagation in that it is more systematic and it uses simpler cost functions. Dependence identification relaxes the constraint that neural activation functions be continuously differentiable, and it determines the number of hidden layers and hidden layer neurons as part of its operation. Dependence identification does not require trial and error with learning rates or network architectures like backpropagation does. Backpropagation performs its learning with a gradient descent of highly nonlinear cost functions. Dependence identification minimizes simple quadratic cost functions which are solved by efficient and well understood methods such as Krylov subspace techniques.

An example has shown that dependence identification may be a good tool for developing a network architecture and set of initial conditions that will subsequently be used by backpropagation. Used this way, DI may be considered as an accelerator for backpropagation, decreasing the development time by eliminating the trial and error search for a proper architecture and developing a set of weights that hopefully avoids many of the possible local minima.
Chapter 6 presents the bound $l_{\text{max}} = \text{ceiling}(\frac{p}{m})n + 1$ (equation (6.2)) as the maximum number of hidden layer neurons created by the dependence identification algorithm for a two layer network. The problem of determining the optimal number of hidden layer neurons is an open question and almost certainly depends on the type of training algorithm employed. Further research will need to be done to determine if there is a correlation between $l_{\text{max}}$ and a possible bound or guideline for the appropriate number of hidden layer neurons for multilayer neural networks in general.

Of the network construction algorithms in chapter 3, dependence identification more closely resembles the neuron addition techniques for constructing boolean networks than the weight decay techniques for pruning continuous networks. However dependence identification is intended to be used for both continuous and boolean network construction. Dependence identification treats groups of linearly dependent patterns analogously to the sets of linearly separable patterns used in boolean network construction. Examples have shown that good results may be obtained for continuous function approximation by using this analogy, however further study is needed to provide a theoretical and analytical justification for the analogy.

The quality and accuracy of the solution determined by dependence identification depends on the sets of $m$ patterns used to create the hidden layers. Obviously if an $m \times m$ portion of the input pattern space is less than full rank then there may be problems, however the procedure for picking optimal or even exceptional sets of $m$ patterns is left as a topic for future research.

Part of dependence identification’s procedure involves the minimizing of quadratic functions, for which block conjugate residual has been proposed. LMS algorithms have appeared in the neural network literature previously as methods for solving the single layer neural network problem [8]. BCR is very fast at minimizing quadratics and it is robust. It would be interesting as a future research goal to compare
minimization with BCR to other LMS minimization procedures.
APPENDIX A

The Makenet Program

This appendix contains the source code for a program which implements the dependence identification algorithm (see algorithm 4). The program is written in C and was compiled on a Sun SPARCstation 2. The program makenet.c appears in section A.3 of this appendix. Section A.2 contains a sample input file for the XOR training problem. The input format is explained in the following section.

A.1 Input Format

The makenet program is called with the following command:

Usage:
makenet [infile [outfile]] [-recompute] [-balk]

The standard input and standard output are used if the input and output files are not specified on the command line. If the -recompute option is used, then the program will recompute the weight matrices of the hidden layer neurons after each complete set of linearly independent patterns is found (see sections 6.3 and 7.2). The -balk option instructs the program to tell the user when a linear system solution failure has occurred (see section 6.4).

The format of the input file is as follows: (see algorithm 4 for an explanation of the variable names).

Line 1 The number of inputs and outputs of the network \(m\) and \(n\). The numbers
are separated by one or more spaces. Remember that one input should be reserved as a constant to create biases on the first layer of neurons.

**Line 2** The number of patterns \( p \) to be taught.

**Line 3** Three error tolerances separated by spaces. The first is the tolerance \( \epsilon_N \), the goal for the total error of the network. The next is \( \epsilon_p \), the tolerance for determining whether patterns are linearly dependent. A third tolerance is also given for determining how accurately the iterative block conjugate residual algorithm (see function `conjres()` in the source code of section A.3) should solve the linear systems. This tolerance should be much smaller than the tolerance \( \epsilon_p \) in order to help prevent “balks.”

**Line 4** This line contains either the word **nonlinear** or **quadratic** depending on which of the two cost functions should be used for determining linear independence and determining whether the overall network has met the tolerance \( \epsilon_N \).

**Line 5** The neural activation function. Possible choices include `sigmoid ( \( \frac{1}{1+e^{-x}} \))`, `tanh`, `step`, `sign` and `linear`. For the discontinuous functions (`step` and `sign`) the word `random` may follow to indicate that when the change of variables \( \Phi(V) = D \) is performed, that random values greater or lesser than zero (depending on the appropriate element of \( D \)) will be used as elements of \( V \). If `random` does not appear then the elements of \( V \) will be either \(-1\) or \(1\). It is also possible to define the output layer as having linear neurons independent of the activation function used in the hidden layers. This is done by adding the words `linear output` to the line. The format of the entire input line is then function `[random] [linear output]`

**Line 6** This line gives the maximum number of layers the network should have \( h_{\text{max}} \).
in algorithm 4).

**p lines** These lines contain the $p \times m$ matrix $U$. Each line contains $m$ input values, and there are $p$ total lines, one for each pattern.

**p lines** These lines contain the $p \times n$ matrix $D$. Each line contains $n$ output values, and there are $p$ total lines, one for each pattern.

Blank lines in the input file are ignored, as well as lines which start with the character “%”. This allows the input file to be commented. An example input file for the XOR problem (see sections 6.1 and 7.1.1) is given on the page.
A.2 Example Input File

% m n:
3 1
% p:
4
% Tolerances (Total error, DI pattern check, and conjugate residual):
0.1 0.4 0.0001
% Indicate whether quadratic or nonlinear error function should be
% used for checking accuracy of network during construction:
linear
% Activation Function. "linear output" may be added for linear o/p neurons.
step
% Maximum number of layers:
2
% U:
0 0 1
0 1 1
1 0 1
1 1 1
% D:
0
1
1
0
0
### A.3 Source Code

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <memory.h>
```

---

The code snippet includes various file inclusion directives and comments that describe the functionalities of the `makenet` program. The program is designed to create a multilayer neural network using the Dependence Identification algorithm and handles various aspects of network creation and modification, such as the ability to assign random positive or negative values to certain network weights, the option to choose between quadratic or nonlinear error functions, and the capacity to discover when a particular group of patterns is not finding any more matches.

---

The source code also includes comments about the creation of the network, the specific types of activations and error tolerances, and the format of the input and output files. Additionally, there's a remark on the issue of singular matrices and the strategy to address it by making them less underspecified.

---

The overall aim is to provide a flexible and robust tool for neural network creation and modification, allowing for various customizations and optimizations to meet specific requirements and challenges in computational tasks.
#include <time.h>
#include <sys/types.h>
#include <sys/param.h>
#include <sys/times.h>

#define PxN(i,j) ((n*(i))+j)
#define PxM(i,j) ((m*(i))+j)
#define PxL(i,j) ((l*(i))+j)
#define MxL(i,j) ((i)+(m*(j)))
#define LxN(i,j) ((i)+(l*(j)))
#define MxN(i,j) ((i)+(m*(j)))

#define TRUE (1)
#define FALSE (0)

#define MAXLINE (16384)
#define LMAX (5000)

#define SIGMIOD(h) (1.0/(1.0+exp(-h)))
#define SIGINV(h) (log(h/(1-h)))
#define STEP(h) (((h) < 0.0)? 0.0 : 1.0)
#define STEPINV(h) (2.0*h - 1.0)
#define SIGN(h) (((h) < 0.0)? -1.0 : 1.0)
#define SIGNINV(h) (h)
#define LIN(h) (h)
#define LININV(h) (h)

#define SIGMIDFNX (0)
#define TANHFNX (1)
#define STEPFNX (2)
#define SIGNFNX (3)
#define LINFNX (4)

#define QUADERR (0)
#define NONLINERR (1)

static struct tms _timestuff;
static long _time1;

#define START_TIMER() do { \
    (void) times(&_timestuff); \
    _time1 = _timestuff.tms_utime; \
} while(0)

#define STOP_TIMER(dtime) do { \
    (void) times(&_timestuff); \
    dtime = (_timestuff.tms_utime) - _time1; \
    dtime /= ((double)HZ); \
} while(0)

#define PRINT_TIMER(dtime) do { \
    (void) fprintf(stderr, "Elapsed time: %4.12e\n", dtime); \

} while(0)

extern float MTMtrace();
extern float evalfnx();
extern float finderror();

/* Global character buffer for input reading */
char ln[MAXLINE];

main(argc,argv)
int argc;
char *argv[];
{
    float *v, *d, *u, **G, tol, ditol, crtol, errd, errv;
    int m, *l,n,p,nlayers,i;
    FILE *inf, *outf;
    double dtime;
    int fnx;
    int errfnx;
    int randv;
    int linop;
    char *s;
    int maxlayers;
    int recompute,nw,nn;
    int saybalk;
    int foundinf;

    /* Default options */
    recompute = FALSE;
    saybalk = FALSE;
    foundinf = FALSE;
    inf = stdin;
    outf = stdout;

    /* Check command line */
    if (argc > 5) usage(argv[0]);
    for (i=1; i<argc; ++i)
        if (!strcmp(argv[i],"-h"))
            usage(argv[0]);
        else if (!strcmp(argv[i],"-recompute"))
            recompute = TRUE;
        else if (!strcmp(argv[i],"-balk"))
            saybalk = TRUE;
        else if (!foundinf) {
            /* Open input file */
            if (!((inf = fopen(argv[i],"r"))){
                fprintf(stderr,
                    "Can not open input training file %s\n",argv[i]);
                usage(argv[0]);
            }
            foundinf = TRUE;
/* Open output file */
} else if ((outf = fopen(argv[i],"w"))) {
    fprintf(stderr,
        "Can not open output weight file %s\n",argv[i]);
    usage(argv[0]);
}

/* Read dimensions: inputs, outputs and number of patterns */
if (!read_line(inf,ln)) {
    BADFILE:
    clearerr(inf);
    fprintf(stderr,"\nTrouble reading input training file\n");
    fprintf(stderr,"Premature end of file\n\n");
    usage(argv[0]);
}
if (!/(sscanf(ln,"%d %d",&m,&n) /=/= 2)/) {
    BADLINE:
    fprintf(stderr,"\nTrouble reading input training file\n");
    fprintf(stderr,"Invalid line\n\n");
    usage(argv[0]);
}
if (!read_line(inf,ln)) goto BADFILE;
if (!sscanf(ln,"%d",&p)) goto BADLINE;

/* Read tolerances */
if (!read_line(inf,ln)) goto BADFILE;
if (sscanf(ln,"%f %f %f",&tol,&ditol,&crtol) /=/= 3) goto BADLINE;

/* Read which error function to use */
if (!read_line(inf,ln)) goto BADFILE;
s = strtok(ln," \t\n");
if (!/(strcmp("quadratic",s))
    errfnx = QUADERR;
else if (!/(strcmp("nonlinear",s))
    errfnx = NONLINERR;
else {
    fprintf(stderr,
        "ERROR: Unrecognized error function (%s) in input file\n",s);
    exit(-1);
}

/* Read Function type */
if (!read_line(inf,ln)) goto BADFILE;
s = strtok(ln," \t\n");
if (!/(strcmp("sigmoid",s))
    fnx = SIGMOIDFNX;
else if (!/(strcmp("tanh",s))
    fnx = TANHFNX;
else if (!/(strcmp("step",s))
    fnx = STEPFNX;
else if (!/(strcmp("sign",s))
fnx = SIGNFNX;
else if (!(strcmp("linear", s)))
    fnx = LINFNX;
else {
    fprintf(stderr,
        "ERROR: Unrecognized sigmoidal function (%s) in input file\n", s);
    exit(-1);
}

/* Check for random V command and linear output */
randv = FALSE;
linop = FALSE;
for (s = strtok(NULL, " \	\n"); s; s = strtok(NULL, " \	\n")) {
    if (!(strcmp("random", s))
        randv = TRUE;
    else if (!(strcmp("linear", s))
        linop = TRUE;
    else if (!(strcmp("output", s))
        /* Do naught */ ;
    else {
        fprintf(stderr,
            "ERROR: Unrecognized option (%s) on function type line!\n", s);
        exit(-1);
    }
}

/* Read maximum number of layers */
if (!read_line(inf, ln)) goto BADFILE;
if (((sscanf(ln, "%d \n", &maxlayers)) || (maxlayers < 1)) {
    fprintf(stderr, "ERROR: Invalid maximum layer value!\n");
    exit(-1);
}

/* Get space for matrices */
if (!((v = (float *)malloc(p*n*sizeof(float))))) {
    NOMEM:
    fprintf(stderr, "ERROR: Can not allocate memory for weight vectors\n");
    exit(-1);
}
if (!((d=(float *)malloc(p*n*sizeof(float)))) goto NOMEM;
if (!((u=(float *)malloc(m*p*sizeof(float)))) goto NOMEM;
if (!((G=(float **)malloc(maxlayers*sizeof(float *)))) goto NOMEM;
if (!((l=(int *)malloc(maxlayers*sizeof(int)))) goto NOMEM;

/* Read matrices from input file */
if (!read_mats(inf, u, d, m, n, p)) usage(argv[0]);
fclose(inf);

/* Compute Solution */
fprintf(stderr, "%d Input/%d Output System with %d Training Patterns\n", m, n, p);
fprintf(stderr, "Network is constrained to have no more than %d layer", 78
```c

maxlayers);
if (maxlayers != 1) fprintf(stderr,"a");
fprintf(stderr,"\nGoal error is \%f\nThe ",tol);
if (errfnx == QUADERR)
    fprintf(stderr,"quadratic");
else fprintf(stderr,"nonlinear");
fprintf(stderr," function is being used for error checking.\n");
if (linop)
    fprintf(stderr,"Output neurons are linear.\n");
fprintf(stderr,"Computing Solution... ");

START_TIMER();
/* Figure out the v matrix */
create_v(v,d,p,n,tol,fnx,randv);
/* Figure out solution */
nlayers = create_net(u,v,d,G,m1,n,p,tol,di tol,cr tol, 
    &errd,&errv,fnx,errfnx,maxlayers,recompute,linop,sa yba lk);
STOP_TIMER(dtime);

if (!nlayers) {
    fclose(outf);
    exit(-1);
}

/* All done */
fprintf(stderr,"Done!\n");
PRINT_TIMER(dtime);
nn = 0;
nw = m*(l[0]-1)+1;
/* Write weight matrices to file */
for (i=0; i<nlayers; ++i) {
    if (i == 0)
        WriteMat(outf,G[i],m1,l[i],i+1);
    else {
        WriteMat(outf,G[i],l[i-1],l[i],i+1);
        if (i < (nlayers - 1))
            nw += l[i-1]*(l[i]-1)+1;
        else nw += l[i-1]*l[i];
    }
    nn += l[i];
}
fclose(outf);
fprintf(stderr,"\n----------------------------------------\n");
if (nlayers == 1)
    fprintf(stderr,"Single Layer Network!\n");
fprintf(stderr,"Network Architecture:\t\%d ",m);
for (i=0; i<(nlayers-1); ++i)
    fprintf(stderr,"- \%d ",l[i]);
fprintf(stderr,"- \%d ",&n);
fprintf(stderr,"Final error (Quadratic):\t\%f ",errv);
fprintf(stderr,"Final error (Nonlinear):\t\%f ",errd);
fprintf(stderr,"Number of layers:\t\%d\n",nlayers);
```

fprintf(stderr,"Number of hidden layer neurons:\t%d\n",nn-n);
fprintf(stderr,"Total number of neurons:\t%d\n",nn);
fprintf(stderr,"Number of weights:\t%d\n",nw);
fprintf(stderr,"Recall that hidden layer constant neurons have\n")
fprintf(stderr,"only one weight connection.\n")
exit(0);
}

/*************************************************************************
*** Function usage() ***
***
*** usage() prints out an error message telling how the program is called
*** when it is called incorrectly or a problem occurs opening a file. The
*** program is then halted.
***
***************************************************************************/

usage(pname)
char *pname;
{
    fprintf(stderr,
        "Usage: %s [infile [outfile]] [-recompute] [-balk]\n\n",pname);
    exit(-1);
}

/*************************************************************************
*** Function read_line() ***
***
*** read_line() reads a line into *ln from the input file *inf. Lines that
*** begin with a '//' and blank lines are ignored. The function returns
*** FALSE when it finds an end of file condition.
***
***************************************************************************/

int read_line(inf,ln)
FILE *inf;
char *ln;
{
    int i;
    int done;

    for (done = FALSE; !done; ) {
        if (feof(inf))
            return(FALSE);
        fgets(ln,MAXLINE,inf);
        /* Check for '//' or all blank line */
        for (i=0; (ln[i]) && (!done); ++i) {

80
if (ln[i] == '%')
    break;

done = ((ln[i] != ' ') && (ln[i] != '\t') && (ln[i] != '\n'));
}
}
return(TRUE);

/*********************************************************/
*** Function read_mats() ***
*** read_mats() reads the input patterns u and the desired output ***
*** patterns v from the input file inf. It returns 1 if successful ***
*** and zero otherwise. ***
*** ***********************************************************/

int read_mats(inf, u, v, m, n, p, tol)
FILE *inf;
float *u, *v;
int m, n, p;
float tol;
{
    int i, j;
    char *s;

    /* Read u */
    for (i = 0; i < p; ++i) {
        if (!read_line(inf, ln)) {
            BADFILE:
                clearerr(inf);
                fprintf(stderr, "\nTrouble reading input file\n");
                fprintf(stderr, "EOF while reading matrices\n\n");
                return(FALSE);
            }
            s = strtok(ln, " \t");
            for (j = 0; j < m; ++j) {
                if (!sscanf(s, "%f", &(u[PxM(i, j)]))) {
                    BADLINE:
                        fprintf(stderr, "\nTrouble reading input file\n");
                        fprintf(stderr, "Invalid line during matric read\n\n");
                        return(FALSE);
                    }
                    s = strtok(NULL, "\t");
                }
            }
    }

    /* Read v */
    for (i = 0; i < p; ++i) {
        if (!read_line(inf, ln)) goto BADFILE;
        s = strtok(ln, " \t");
    }

78
for (j = 0; j < n; ++j) {
    if (!sscanf(s,"%f",&(v[PxN(i,j)]))) goto BADLINE;
    s = strtok(NULL, "\t");
}
}

return(TRUE);

/*****************************************************************************/

*** Function conjres() ***
*** conjres() uses a block conjugate residual routine to calculate ***
*** the weight matrix w which minimizes ***
*** trace((v - u*w)'*(v - u*w)) ***
*** where V is p X n, U is p X m, and the solution w is m X n. ***
*** The determination of w is halted when the given tolerance tol is ***
*** reached. ***
*** It returns TRUE if it is successful and FALSE if it is not. ***
****************************************************************************/

int conjres(w,u,v,m,n,p,tol)
float *w, *u, *v;
int m, n, p;
float tol;
{
    float *r, *q, alpha, beta, ad, e, de, *x, *y, *z;
    int i, j;

    /* Get space for matrices */
    if (!((r = (float *)malloc(p*n*sizeof(float))))) {
        NOMEM:
        fprintf(stderr,"\nERROR: Can not allocate memory for matrix\n");
        return(FALSE);
    }
    if (!((q = (float *)malloc(m*n*sizeof(float))))) goto NOMEM;
    if (!((x = (float *)malloc(p*n*sizeof(float))))) goto NOMEM;
    if (!((y = (float *)malloc(m*n*sizeof(float))))) goto NOMEM;
    if (!((z = (float *)malloc(p*n*sizeof(float))))) goto NOMEM;

    /* Calculate initial residual */
    MMmult(x,1.0,u,w,p,m,n);
    MMadd(r,v,-1.0,x,p,n);

    /* Calculate initial Search direction P */
    MTMmult(q,1.0,u,r,p,m,n);

    /* Calculate x = u*q and denominator of alpha (ad) */
    MMmult(x,1.0,u,q,p,m,n);
    ad = MTMtrace(x,x,p,n);
/ Calculate initial error */
e = MTMtrace(r,r,p,n);
de = 10.0*tol;

/* Start iterative loop */
while ((e > tol) && (de > tol)) {
    /* Compute alpha */
    alpha = MTMtrace(x,r,p,n);
    alpha /= ad;
    
    /* Compute new weights */
    MMadd_c(w,w,alpha,q,m,n);
    
    /* Compute new residual */
    MMadd(r,r,-alpha,x,p,n);
    
    /* Compute new error */
    de = e;
    e = MTMtrace(r,r,p,n);
    de = fabs(de-e);
    
    /* Compute beta */
    MTMmult(y,1.0,u,x,p,m,n);
    MTMmult(z,1.0,u,y,p,m,n);
    beta = MTMtrace(z,x,p,n)/ad;
    
    /* Compute new search direction q */
    MMadd_c(q,y,-beta,q,m,n);
    
    /* Compute new denominator of alpha (ad) */
    MTMmult(x,1.0,u,q,p,m,n);
    ad = MTMtrace(x,x,p,n);
}

/* All done */
free(z);
free(y);
free(x);
free(q);
free(r);
return(TRUE);

/*****************************/

*** Function MTMmult()  ***
***                        ***
*** MTMmult() computes      ***
***     q = s*u'*r          ***
**where s is a scalar,**

**' stands for transpose,**

**u is an p X m matrix,**

**r is an p X n matrix,**

**and the computed matrix q is m X n.**

```c
MTMmult(q, s, u, r, p, m, n)
float *q, *s, *u, *r;
int p, m, n;
{
    int i, j, k;
    for (i = 0; i < m; ++i)
        for (j = 0; j < n; ++j) {
            q[MxN(i, j)] = 0.0;
            for (k = 0; k < p; ++k)
                q[MxN(i, j)] += u[PxM(k, i)]*r[PxN(k, j)];
        }
}
```

**Function MMadd()**

**MMadd() computes**

**where r, v and x are p X n matrices and s is a scalar.**

```c
MMadd(r, v, s, x, p, n)
float *r, *v, *s, *x;
int p, n;
{
    int i, j;
    for (i = 0; i < p; ++i)
        for (j = 0; j < n; ++j)
            r[PxN(i, j)] = v[PxN(i, j)] + s*x[PxN(i, j)];
}
```

**Function MMadd_c()**

**MMadd() computes**

**where r, v and x are m X n matrices and s is a scalar.**

**MMadd_r() is used when the matrices are stored in column order instead of row order like normal matrices.**
MMadd_c(r, v, s, x, m, n)
float *r, *v, s, *x;
int m, n;
{
    int i, j;
    for (i=0; i<m; ++i)
        for (j=0; j<n; ++j)
            r[MxN(i, j)] = v[MxN(i, j)] + s*x[MxN(i, j)];
}

MMmult(x, s, u, w, p, m, n)
float *x, *s, *u, *w;
int p, m, n;
{
    int i, j, k;
    for (i=0; i<p; ++i)
        for (j=0; j<n; ++j) {
            x[PxN(i, j)] = 0.0;
            for (k = 0; k<m; ++k)
                x[PxN(i, j)] += u[PxM(i, k)]*w[MxN(k, j)];
        }
}

MTMtrace(x, y, p, n)
float MTMtrace(x, y, p, n)
float *x, *y;
int p, n;
{
    int i, k;
    float ad;

    ad = 0.0;
    for (i = 0; i < n; ++i)
        for (k = 0; k < p; ++k)
            ad += x[PxN(k, i)] * y[PxN(k, i)];

    return(ad);
}

/******************************************************************************
 *** Function WriteMat() ***
 *** WriteMat() writes the matrix w to the file outf in a format readable by MatLab. The MatLab variable name for the matrix is the string mname. The dimensions of the matrix are m X n. ***

WriteMat(outf, w, m, n, mnum)
FILE *outf;
float *w;
int m, n;
int mnum;
{
    int i, j;

    /* Write out w matrix */
    fprintf(outf, "W%d = [%n", mnum);
    for (i = 0; i < m; ++i) {
        for (j = 0; j < n; ++j)
            fprintf(outf, "%f ", w[MxN(i, j)]);
        fprintf(outf, "\n");
    }
    fprintf(outf, "];\n");
}

/******************************************************************************
 *** Function create_v() ***
 *** create_v() computes the elements of the p X n matrix v which are the inverses of the elements of d, corresponding to the appropriate fnx number. ***

create_v(outf, w, m, n, mnnum)
FILE *outf;
float *w;
int m, n;
imnum;
{
    int i, j;

    /* Write out v matrix */
    fprintf(outf, "W%d = [%n", mnum);
    for (i = 0; i < m; ++i) {
        for (j = 0; j < n; ++j)
            fprintf(outf, "%f ", w[MxN(i, j)]);
        fprintf(outf, "\n");
    }
    fprintf(outf, "];\n");
}
create_v(v,d,p,n,tol,fnx,randv)
float *v,*d;
int p,n;
float tol;
int fnx,randv;
{
    int i,j;
    float x,maxran;
    long rv;

    maxran = (float)(0x7fffffff);
    if (!randv) x = 1.0;
    tol = tol/((float)p);
    if (tol < 0.0001)
        tol = 0.0001;

    for (i = 0; i < p; ++i) {
        for (j = 0; j < n; ++j) {
            switch (fnx) {
            case SIGMOIDFNX:
                if ((1.0 - d[PxN(i,j)]) < tol/2.0)
                    x = 1.0-tol/2.0;
                else if (d[PxN(i,j)] < tol/2.0)
                    x = tol/2.0;
                else x = d[PxN(i,j)];
                v[PxN(i,j)] = SIGINV(x);
                break;
            case TANHFNX:
                if (((1.0 - fabs(d[PxN(i,j)]))) < tol/2.0)
                    x = SIGN(d[PxN(i,j)])*(1.0-tol/2.0);
                else x = d[PxN(i,j)];
                v[PxN(i,j)] = atanh(x);
                break;
            case STEPFNX:
                if (randv)
                    x = 2.0*(float)random()/maxran+0.25;
                v[PxN(i,j)] = STEPINV(d[PxN(i,j)])*x;
                break;
            case SIGNFNX:
                if (randv)
                    x = 2.0*(float)random()/maxran+0.25;
                v[PxN(i,j)] = SIGNINV(d[PxN(i,j)])*x;
                break;
            case LINFNX:
                v[PxN(i,j)] = LININV(d[PxN(i,j)]);
                break;
            }
        }
    }
}
/** Function create_net()  
***  
*** create_net() computes the neural network. It is constrained to  
*** have no more layers than the value maxlayers. It returns the  
*** actual number of layers. If there is a problem, it returns 0.  
***  
*********  
*/

int create_net(u,v,d,G,m,ll,n,p,tol,ditol,crtol,errd,errv,
    fnx,errfnx,maxlayers,recompute,linop,saybalk)
float *u, *v, *d, **G;
int m, *ll, n, p;
float tol,ditol,crtol,*errd,*errv;
int fnx, errfnx, maxlayers, recompute,linop,saybalk;
{
    int i,j,k,l,I,mm,*Dep,miss,found,lfound,pr,dn,mm;
    int Tf;
    int nlay;

    /* Check # of patterns in training set */
    if (m > p) {
        fprintf(stderr,"Number of patterns should be greater than\n");
        fprintf(stderr,"or equal to the number of inputs!\n");
        return(0);
    }

    /* Create dependence matrix */
    if ((!Dep = (int *)malloc(p*sizeof(int)))) {
        NOMEM:
        fprintf(stderr,"\nERROR: Can not allocate memory for matrix\n");
        return(0);
    }

    /* Hidden layer activations and error initialization: */
    H = NULL;

    /* Look at D or V depending on error function and output activation fnx */
    if (linop)
        op1 = d;
    else
        op1 = v;
    if (errfnx == NONLINERR) {
        op2 = d;
        err = errd;
    } else {
        op2 = v;
        err = errv;
    }

    /* Loop until we have maximum # of layers or we've solved problem */
for (nlay = 0; (TRUE); ++nlay)
{
    /* Create this layer's weight matrix */
    if (! (G[nlay] = (float *)malloc(m*LMAX*sizeof(float)))) goto NOMEM;

    /* Initialize weight matrix to zeros */
    memset((char *) G[nlay], 0, m*n*sizeof(float));

    /* Determine weights for entire input space using conjugate residual */
    if (! conjres(G[nlay], u, op1, m, n, p, crtol)) {
        fprintf("Problem with full input conjugate residual!
"");
        return(0);
    }

    /* Compute error of solution, exit if it meets tolerance */
    /* or we've computed the maximum number of layers */
    if (! (out = (float *)malloc(p*n*sizeof(float)))) goto NOMEM;
    if (! (*err = finderror(out, d, v, u, G[nlay], p, n, m, fnx, errfnx)) <= tol) |
        (nlay == (maxlayers-1))
        break;

    /* Create necessary matrices */
    if (recompute) {
        if (! (ur = (float *)malloc(p*m*sizeof(float)))) goto NOMEM;
        if (! (vr = (float *)malloc(p*n*sizeof(float)))) goto NOMEM;
    }
    if (! (uu = (float *)malloc(m*m*sizeof(float)))) goto NOMEM;
    if (! (vv = (float *)malloc(m*n*sizeof(float)))) goto NOMEM;
    if (! (g = (float *)malloc(m*n*sizeof(float)))) goto NOMEM;
    if (! (T = (float *)malloc(n*sizeof(float)))) goto NOMEM;

    /* Initialize # of hidden layer units and single layer values */
    ll[nlay] = 0;
    l = 0;
    miss = m;

    /* Initialize Dep vector showing groups of independent patterns */
    memset((char *) Dep, 0, p*sizeof(int));
    dn = 0;

    /* Loop through, finding hidden layer weights */
    I = 0;
    lfound = p;
    mn = 0;
    for (found = 0; ((found < p) && (l < (LMAX - n))); ) {
        /* Create a training subset */
        for (k = 0; k < miss; ++k) {
            /* Find an independent pattern */
            while (Dep[I] != dn) {
                ++I;
                if (I >= p) I = 0;
            }
memcpy((char *)&uu[PzM(k,0)], (char *)&u[PzM(I,0)], m*sizeof(float));
memcpy((char *)&vv[PzN(k,0)], (char *)&v[PzN(I,0)], n*sizeof(float));
++I;
if (I >= p) I = 0;
}

/* Augment independent patterns with dependent ones */
if (!found != found) {
    I -= miss + 1;
    if (I < 0) I += p;
    for (k = miss; k < m; ++k) {
        while (Dep[I] == dn) {
            --I;
            if (I < 0) I = p - 1;
        }
    }
    memcpy((char *)&uu[PzM(k,0)], (char *)&u[PzM(I,0)], m*sizeof(float));
    memcpy((char *)&vv[PzN(k,0)], (char *)&v[PzN(I,0)], n*sizeof(float));
    --I;
    if (I < 0) I = p - 1;
}
mm = m;
else mm = miss;

/* Initialize g to zeros */
memb((char *)g, 0, mm*n*sizeof(float));
/* Determine weights for the pattern subset */
if (!conjres(g, uu, vv, m, n, mm, crtol)) {
    fprintf(stderr, "Problem with conjugate residual!\n");
    return(0);
}

/* DEBUGGING STUFF: */
/**************************
fprintf(stderr,"Trying uu =\n");
for (i=0; i<mm; ++i) {
    for (j=0; j<m; ++j)
        fprintf(stderr,"%4.4f ", uu[PzM(i, j)]);
    fprintf(stderr,"\n");
}
fprintf(stderr,"vv =\n");
for (i=0; i<mm; ++i) {
    for (j=0; j<n; ++j)
        fprintf(stderr,"%4.4f ", vv[PzN(i, j)]);
    fprintf(stderr,"\n");
}
/******************************
/* Compute output and mark I/O patterns which are independent */

90
pr = 0;
miss = 0;
lfound = found;
for (i = 0; i < p; ++i) {
    if (Dep[i] == dn) {
        Tf = FALSE;
        MMmult(T, 1.0, &u[PxM(i, 0)], g, 1, m, n);
        for (j = 0; j < n; ++j) {
            if (erfnx == NONLINERR)
                T[j] = evalfnx(T[j], fnx);
            if ((fabs(T[j] - op2[PxN(i, j)])) > ditol) {
                ++miss;
                ++Dep[i];
                Tf = TRUE;
                break;
            }
        }
        if (!Tf) {
            ++found;
            /* Create overall dependent training set */
            if (recompute) {
                memcpy(&ur[PxM(pr, 0)], &u[PxM(i, 0)], m * sizeof(float));
                memcpy(&vr[PxN(pr++, 0)], &v[PxN(i, 0)], n * sizeof(float));
            }
        }
    }
}
/* Number of patterns in next training set: */
if (miss > m)
    miss = m;
++dn;
if (!lfound != found) {
    /* Recompute g if necessary */
    if (recompute && (pr > m))
        if (!conjres(g, ur, vr, m, n, pr, crtol)) {
            fprintf(stderr, "Problem with conjugate residual on weight matrix recomputation!\n");
            return(0);
        }
    /* Copy g into big G matrix */
    memcpy(&G[nlay][MxL(0, 1)], g, m * sizeof(float));
    l += n;
    mn = 0;
} else {
    if (saybalk) fprintf(stderr, "BALK\n");
    if (miss + found == p) {
        if (saybalk) fprintf(stderr, "Big BALK!\n");
        ++mn;
        miss = miss - mn;
    } else {

if (miss != 0) {
    if (saybalk)
        fprintf(stderr,"Biggest Balk!
\n");
    /* Copy g into big G matrix anyway */
    memcpy(&(G[nlay][MxL(0,1)]), g,
           m*n*sizeof(float));
    l += n;
    break;
}
/* End for */

/* Add a threshold neuron in the hidden layer */
++l;
for (i = 0; i < m-1; ++i)
    G[nlay][MxL(i,1)] = 0.0;
G[nlay][MxL(m-1,1-1)] = 1.0;

/* Compute hidden layer activations, set them equal to new U */
if (H) free(H);
if (!((H = (float *)malloc(p*l*sizeof(float)))) goto NOMEM;
findH(H, u, G[nlay], p, m, l, fnx);
u = H;

/* Free temporary matrices and resize */
if (recompute) {
    free(ur);
    free(vr);
}
free(uu);
free(vv);
free(g);
free(T);
free(out);
G[nlay] = (float *)realloc(G[nlay], m*l*sizeof(float));

/* Keep track of dimensions */
l[nlay] = l;
m = l;
}
/* END outer for loop for layers */

/* Resize the current G matrix */
G[nlay] = (float *)realloc(G[nlay], m*l*sizeof(float));

/* Compute the error that was not used by the algorithm */
if (errfnx == NONLINERR)
    *errv = finderror(out, d, v, u, G[nlay], p, n, m, fnx, QUADERR);
else *errd = finderror(out, d, v, u, G[nlay], p, n, m, fnx, NONLINERR);
/* Set the number of neurons in the output layer and exit. */
ll[nlay++] = n;
return(nlay);

/* Function evalfnx() */
/* evalfnx() evaluates y depending on the appropriate activation fnx */
float evalfnx(y,fnx)
float y;
int fnx;
{
    switch (fnx) {
        case SIGMOIDFNX: return(SIGMOID(y));
        case TANHFNX: return(tanh(y));
        case STEPFNX: return(STEP(y));
        case SIGNFNX: return(SIGN(y));
        case LINFNX: return(LIN(y));
    }
}

/* Function findH() */
/* findH() computes the activations of the hidden layer neurons. */
findH(H,u,G,p,m,l,fnx)
float *H,*u,*G;
int p,m,l,fnx;
{
    int i,j,k;
    for (i=0; i<p; ++i) {
        for (j=0; j<l; ++j) {
            H[PxL(i,j)] = 0.0;
            for (k = 0; k < m; ++k)
                H[PxL(i,j)] += u[PxM(i,k)]*G[MxL(k,j)];
            H[PxL(i,j)] = evalfnx(H[PxL(i,j)],fnx);
        }
    }
}

/* Function finderror() */
finderror() computes the total mean square error of the network.

```c
float finderror(out, d, v, H, W, p, n, l, fnx, errfnx)
int p, n, l, fnx, errfnx;
{
    int i, j, k;

    for (i = 0; i < p; ++i)
        for (j = 0; j < n; ++j) {
            out[PxN(i, j)] = 0.0;
            for (k = 0; k < l; ++k)
                out[PxN(i, j)] += H[PxL(i, k)] * W[LxN(k, j)];
            switch(errfnx) {
                case NONLINERR: out[PxN(i, j)] =
                    d[PxN(i, j)] - evalfnx(out[PxN(i, j)], fnx);
                    break;
                case QUADERR: out[PxN(i, j)] =
                    v[PxN(i, j)] - out[PxN(i, j)];
                    break;
            }
        }
    return(MTMtrace(out, out, p, n));
}
```
BIBLIOGRAPHY


