AFIT/GEO/ENG/91D-02

FUNCTION PREDICTION USING RECURRENT NEURAL NETWORKS

THESIS

Randall L. Lindsey
Captain

AFIT/GEO/ENG/91D-02

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
FUNCTION PREDICTION USING RECURRENT NEURAL NETWORKS

5. FUNDING NUMBERS

6. AUTHOR(S)
Randall L. Lindsey, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GEO/ENG/91D-02

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Mike Hinman
RADC/IRR
Griffis AFB NY 13441

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A fully recurrent neural network was applied to the function prediction problem. The real-time recurrent learning (RTRL) algorithm was modified and tested for use as a viable function predictor. The modification gave the algorithm a variable learning rate and a linear/sigmoidal output selection. Verifying the networks ability to temporally learn both the classic exclusive-OR (XOR) problem and the internal state problem, the network was then used to simulate the frequency response of a second order IIR lowpass Butterworth filter. The recurrent network was then applied to two problems: head position tracking, and voice data reconstruction. The accuracy at which the network predicted the pilot's head position was compared to the best linear statistical prediction algorithm. The application of the network to the reconstruction of voice data showed the recurrent network's ability to learn temporally encoded sequences, and make decisions as to whether or not a speech signal sample was considered a fricative or a voiced portion of speech.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Recurrent Neural Networks, Real-time Recurrent Learning Algorithm, Function Prediction, Recurrent Network Applications, Neural Nets, Recurrent Backpropagation	110
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements.***

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in... . When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

AFIT/GEO/ENG/91D-02

FUNCTION PREDICTION
USING RECURRENT NEURAL NETWORKS

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Randall L. Lindsey, B.S.E.E.
Captain

December, 1991

Acknowledgments

The last thing I wanted to research at AFIT was any topic dealing with neural networks. Yet, after seeing the utility of neural networks in solving many practical applications, my view of neural network research changed drastically. Thanks to the initiative of my thesis advisor, Capt Dennis Ruck, and the insight of my committee members, Maj Steven Rogers and Dr Matthew Kabrisky, I was able to research the specific area of neural networks that interested me the most: recurrent neural networks. I would never have conceived of this research topic, let alone attempted it, without their guidance and direction. In addition, many thanks belong to by faithful wife, Linda, whose support and understanding kept me coming back to school with a smile on my face and a song in my heart. Finally, it was my faith in Jesus Christ that helped me see the light at the end of the tunnel. With this in mind, I was always able to keep a balanced perspective between school and my family life. As important as this education is in my career, I would have dropped it in a second if it meant compromising my faith or my family. Thank you Jesus for giving me the strength to carry this commitment through.

Randall L. Lindsey

Table of Contents

List of Figures

List of Tables

Abstract

A fully recurrent neural network was applied to the function prediction problem. The real-time recurrent learning (RTRL) algorithm was modified and tested for use as a viable function predictor. The modification gave the algorithm a variable learning rate and a linear/sigmoidal output selection. Verifying the networks ability to temporally learn both the classic exclusive-OR (XOR) problem and the internal state problem, the network was then used to simulate the frequency response of a second order IIR lowpass Butterworth filter. The recurrent network was then applied to two problems: head position tracking, and voice data reconstruction. The accuracy at which the network predicted the pilot's head position was compared to the best linear statistical prediction algorithm. The application of the network to the reconstruction of voice data showed the recurrent network's ability to learn temporally encoded sequences, and make decisions as to whether or not a speech signal sample was considered a fricative or a voiced portion of speech.

FUNCTION PREDICTION
USING RECURRENT NEURAL NETWORKS

I. Introduction

The ability of machines to perform accurate function prediction remains an unsolved problem. Although conventional sensors used in military applications provide enough information for a human to predict an event's outcome, the extension to automatic prediction by machines is still impractical using current computer architectures. According to Webster (8), to predict is to

> declare in advance; esp: foretell on the basis of observation, experience, or scientific reason.

Therefore, function prediction, as defined in this thesis, is the declaration of the future value of a specific function based upon that function's history.

The use of recurrent neural network theory provides a novel approach to solving this problem. Biological neural networks readily and easily process temporal information; artificial neural networks should do the same. Formulated from biological research, artificial neural networks provide a unique approach to solving problems that could prove quite successful in the areas of speech processing, image recognition, and function prediction (7). Recurrent neural networks are artificial neural networks which permit the encoding and learning of temporal sequences. This is an important feature in a world governed by time dependent processes. Thus, properly trained recurrent neural networks could prove quite successful in applications involving time dependencies, including function prediction.

1.1 Problem

The goal of this thesis is to perform accurate function prediction using recurrent neural networks.

1.2 Background

Publications in the field of neural networks span a multi-disciplinary spectrum: neurobiology, physics, psychology, medical science, mathematics, computer science, and engineering. As such, it is difficult to accurately compile a thorough summary of where neural network technology stands today. However, a broad sampling of current literature centered on the topic of recurrent backpropagation neural networks yields a more focused review. Chapter II contains highlights of some of the most promising recurrent neural network algorithms, namely backpropagation through time (BPTT), modified BPTT, real-time recurrent learning (RTRL), and subgrouped RTRL.

As technology improves, new and innovative algorithms are discovered which help researchers and engineers alike in solving time-dependent problems. All of the algorithms previously discussed possess the ability, if properly trained, to tackle many difficult temporal tasks. Several of these algorithms are simply modifications of the backpropagation through time method, or the real-time recurrent learning method.

1.3 Assumptions

It is assumed that the input feature vectors have already been selected for use in training and testing the network.

1.4 Scope

The scope of this thesis will focus on solving the function prediction problem using recurrent neural network theory. This theory is based on a modification of the RTRL algorithm (23).

1.5 Approach

Function prediction using recurrent neural networks will be accomplished in four steps. First, the recurrent neural network program must be created. The RTRL algorithm will be coded using the C programming language. It will be tested using several temporally encoded data sets to verify its performance. Second, the network's output will be modified to determine if linear outputs combined with sigmoidal "hidden units" (processing units which have no external connections) will further optimize the network's response. This modification will enable the network to predict unbounded functions. Third, a variable learning rate will be added to the network training algorithm to enhance the rate of convergence. Finally, several functions will be used to test the network's prediction abilities, including two specific applications.

II. Literature Review

2.1 Introduction

In this literature review, the current state of recurrent neural network technology is summarized.

Biological neural networks readily and easily process temporal information; artificial neural networks should do the same. Formulated from biological research, artificial neural networks provide a heuristic approach to solving problems that could prove quite successful in the areas of speech processing and image recognition (7). Recurrent neural networks are artificial neural networks which use feedback to encode and learn temporal sequences. This is an important feature in a world governed by time dependent processes. Thus, properly trained recurrent neural networks could prove quite successful in applications involving time dependencies.

This section contains a short background on basic neural network theory to aid the reader's comprehension of that subject. In addition, the following algorithms are highlighted: backpropagation through time (BPTT), modified BPTT, real-time recurrent learning (RTRL), and subgrouped RTRL. These algorithms summarize the current improvements in recurrent backpropagation neural network technology.

2.2 Background

Artificial neural networks are nothing more than an application of biological concepts to electronic machines. Another name for an artificial neural network is a neuromime. It is called a neuromime because it attempts to copy or mimic the response of a true biological neuron, the most basic processing element of the brain (13).

During the late 1950's, Rosenblatt invented a new class of machines which seemed to offer what many researchers thought was a natural and powerful model of machine learning (15). It was called the perceptron. The basic perceptron model consists of an

array of input sensory nodes randomly connected to a second array of associative nodes. The random connections are called weights. The weights are randomly generated values in the range [-1,1]. Each of the secondary nodes produces an output only if enough of the sensory nodes connected to it are activated. The sensory nodes can be viewed as the means by which outside information is captured by the machine, and the associative nodes can be viewed as the input to the machine.

The output, or response, of the perceptron is proportional to the weighted sum of the associative node responses. In other words, if x_i denotes the response of the ith associative node and w_i denotes the corresponding connection weight, then the response is given by

$$R_n = \sum_{i=1}^{n} w_i x_i$$

where n is the total number of associative nodes. Thus for a positive R, the stimulus is said to belong to class 1, and for a negative R, the stimulus is said to belong to class 2. That is how a decision is made. In its most basic form, the basic perceptron is simply an implementation of a linear decision function. The perceptron learns by changing the connection weights in such a way as to minimize the total response error. The nodal error is the difference between the desired output and the actual computed response of the node. In equation form, the error is given by

$$e_n = d_n - R_n$$

where e_n is the error of node n, and d_n is the desired value of node n. Therefore, the total response error is the summation of the nodal errors over the entire length of the data set (epoch).

In most applications, the output of the network is processed by a differentiable function, usually the logistic squashing function (sigmoid). The output of the sigmoid is

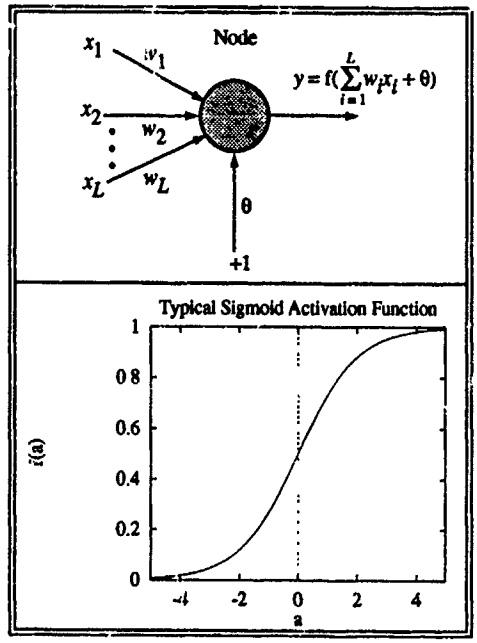


Figure 1. Single-layer perceptron with sigmoidal processing. The output of the node is the weighted sum of the inputs, processed through the sigmoid function. The sigmoid function is displayed in the lower part of the figure.

given by

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

When the network input is processed by this function, the response is the weighted sum of the inputs, including the bias term θ, processed through the sigmoid function. Thus the resulting output is given by

$$R_n = f_n(\sum_{i=1}^{n} w_i x_i + \theta)$$

Figure 1 details the output of the sigmoid and gives a good picture of what the network node should be viewed as.

To date, many other architectures have been proposed which extend the basic concepts introduced by Rosenblatt. These new networks are called by various names: mul-

tilayer perceptrons, feedforward neural networks, backpropagation networks, recurrent backpropagation, and so on. The term backpropagation refers to the way interconnection weights are updated; that is by propagating backward from the output to the input, changing each connection weight in such a way as to minimize the total error. A recurrently connected neural network is a backpropagation network that contains feedback loops from previous states (timed inputs). The outputs that feedback are used as part of the next sequentially timed input. So, the output at time $t + 1$ is predictive based upon the current input and the previous output. As with the input vector, the feedback connections each have their own adaptable weights. These recurrent weights are changed just as before in order to minimize the total error over the epoch length. Figure 2 shows a general layout of a recurrent neural network. Notice that the current input vector at time t is composed of a bias (always equal to one), the external inputs, and the previous network's output. This is a convenient way to show how feedback is processed through the network.

2.3 Scope

Publications in the field of neural networks span a multi-disciplinary spectrum: neurobiology, physics, psychology, medical science, mathematics, computer science, and engineering. As such, it is difficult to accurately compile a thorough summary of where neural network technology stands today. However, a broad sampling of current literature centered on the topic of recurrent backpropagation neural networks yields a more focused review.

The scope of this review will focus on current literature detailing the improvements in recurrent backpropagation neural network technology. Most of the improvements presented are simply modified versions of previously published work.

2.4 Backpropagation through time (BPTT)

Much of the current research has focused on the use of recurrent neural networks that deal with time-varying input or output in nontrivial ways. Rumelhart describes a general

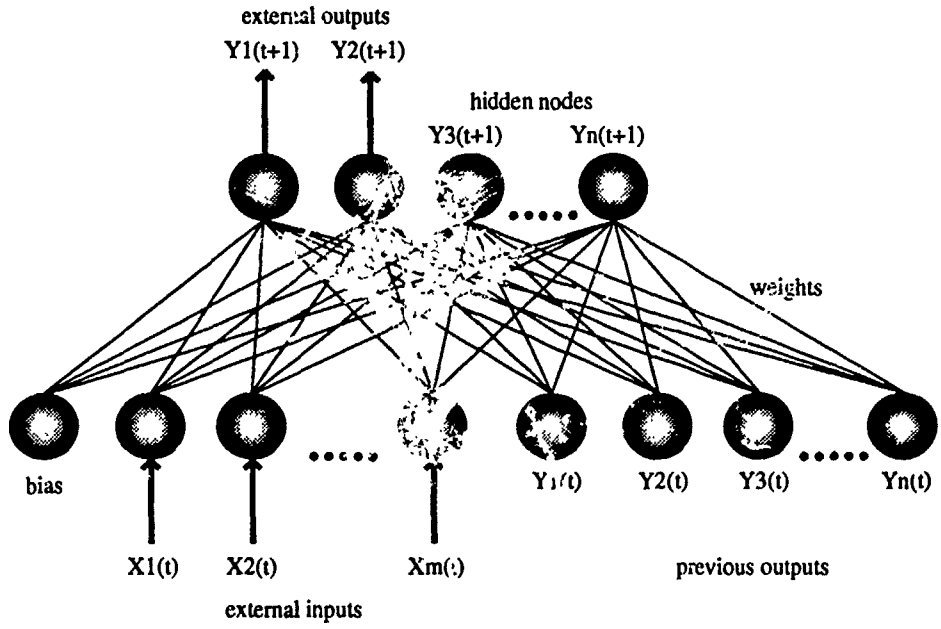


Figure 2. A general recurrent neural network. The number of external inputs, external outputs and nodes are user-defined. The output at time $t + 1$ is predictive based upon the current input and the previous output. The network is fully interconnected by connection weights, adjusted using the gradient-descent method. Feedback is introduced by using the network's previous output as part of the current input.

framework for such a problem as a recurrent network which unfolds into a multilayer feedforward network that grows by one layer on each time step (18). The adjustments to the network's connection weights are designed to minimize a time-averaged measure of the network's overall learning error. This is referred to as backpropagation through time (BPTT). Its strength lies in its generality, but a corresponding weakness is its growing memory requirement when trained on arbitrarily long sequences. It is this method (BPTT) which most researchers tend to modify. For example, Rohwer and Forrest (14) presented a variation of the backpropagation feedforward training method (18). This method can be indirectly applied to time-dependent problems in arbitrarily connected networks by modeling a virtual network made from several copies of the original, with one copy for each time step. The adjustments to the network's connection weights are designed

to minimize a time-averaged measure of the network's overall learning error. In this method, errors are assessed and handled simultaneously throughout the network rather than propagated through it. It can be applied directly to arbitrarily connected networks, provided that a certain criterion related to the training problem is satisfied. When this criterion is not met, a modification of the training problem can be found which properly improves the stability of the network.

...5 *Modified BPTT*

There are many recurrent neural network models whose architectures are modified versions of previously published work. For example, Pineda (10) has recently generalized Rumelhart's backpropagation learning algorithm for feedforward neural networks (18) to recurrent neural networks. Pearlmutter (9) has further generalized this algorithm to recurrent networks that produce time-dependent trajectories. The Pearlmutter architecture requires much more training time than that of the Rumelhart or Pineda algorithms. As a result, Fang and Sejnowski (2) modified the Pearlmutter algorithm to improve both its performance and speed. The Fang-Sejnowski article detailed the modifications on the learning update rule which allows adaptable independent learning rates for individual parameters in the algorithm. This allows fast parameter estimation while avoiding most cases of catastrophic divergences.

2.6 Real-Time Recurrent Learning (RTRL)

One particularly interesting article describes a learning algorithm for training completely recurrent, continually updated networks to learn temporal tasks (23). This technique emphasizes using uniform starting configurations that contain no previously known information about the temporal nature of the task. More precisely, it is a gradient-following learning algorithm which tracks the total network error along a trajectory which minimizes this total error. Its main advantage is that it does not require a precisely defined training interval. It operates while the system is running. A disadvantage is that it requires nonlo-

cal communication during training. This means it is computationally expensive. Yet, the algorithm allows recurrently connected networks to learn complex tasks that require the retention of information over fixed or indefinite time periods. This algorithm is referred to as the real time recurrent learning (RTRL) algorithm. It is this algorithm that this thesis effort is based upon.

2.7 Subgrouped RTRL

Whereas RTRL has been shown to have great power and generality, it has the disadvantage of requiring a great deal of computation time (CPU intensive). To address this problem, Zipser proposed an improved technique which reduces the amount of computation required by RTRL without changes in network connectivity (24). The reduction in computation time is a result of network subgrouping. The original network is divided into subnets for the purpose of error propagation, leaving them undivided for activity propagation. This means that during training, the network is subgrouped only when the error is propagated backward through the network's connection weights. During the normal feedforward propagation portion, the network remains fully connected. A comparison of this new method and the previous RTRL method showed the subgrouped RTRL algorithm to be 10 times faster on learning to be a finite-state part of a Turing machine (24).

2.8 Summary

As technology improves, new and innovative algorithms are discovered which help researchers and engineers alike in solving time-dependent problems. All of the algorithms previously discussed possess the ability, if properly trained, to tackle or completely solve many difficult temporal tasks. Several of these algorithms are simply modifications of the backpropagation through time method, or the real time recurrent learning method.

Although great strides have been made in advancing recurrent neural network technology, further research is still needed. Most of these algorithms are implemented on digital machines. Because of this, routines can be constructed in code which cannot be

physically realizable. That is, they cannot be implemented in hardware configurations. Therefore, further research is necessary to determine whether or not these recurrently connected networks can be realized as physical elements, thus greatly increasing their speed and utility.

III. Methodology

3.1 Introduction

Citing the work of several neural network researchers, Chapter II covered a subset of the most recent research into recurrent artificial neural network algorithms. Specifically noted were the real-time recurrent learning (RTRL) algorithm, and the subgrouped RTRL algorithm. This thesis seeks to encode the RTRL algorithm, perform several modifications, and use the resulting network as a reliable engine for function prediction problems.

This chapter covers the development, modifications, and testing of the RTRL algorithm for function prediction applications at AFIT. The basics of the RTRL algorithm along with the current modifications of this algorithm are described. In addition, the testing procedures and training methods used on the algorithm are discussed. The chapter concludes with a description of how the recurrent neural network was applied to two specific problems.

3.2 RTRL Algorithm

The real-time recurrent learning algorithm (23) is a gradient-following algorithm for completely recurrent networks running in continually sampled time. The architecture of the network consists of a user specified number of input nodes, a unity input bias, and a user specified number of "hidden nodes" and output nodes (see Figure 2). The output nodes (or hidden nodes for that matter) were designed originally to process the nodal activation using the sigmoid function (Eq 1). The nodal activation is defined as the weighted sum of all the inputs to a particular processing node. Each output node is fully connected, by weighted connections, to every other node in the network, including external inputs and previous outputs. Once the output is computed, it is returned and used as a part of the new network input.

The derivation of the RTRL algorithm is contained in the article by Williams and Zipser (23). In this thesis, only the most important equations will be highlighted. The basic network has n units (nodes) and m external inputs. Any or all of the network units can be outputs. Let $y_k(t)$ denote the output of the kth node at time t, and let $x_k(t)$ denote the kth external input signal to the network at time t. Now define $z_k(t)$ to be the composite network input at time t. In other words, $z_k(t)$ is obtained by concatenating $x(t)$ and $y(t)$, so that

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \tag{2}$$

where U denotes the set of indices k such that z_k is the output of a unit in the network, and where I denotes the set of indices k such that z_k is an external input. Note: the unity bias term is assumed to be a part of the m inputs. With a fully interconnected network, the weight matrix w_{ij} becomes a single $n \times (m+n)$ matrix, with i corresponding to a specific node, and j corresponding to a specific input.

The output of the kth unit as a function of the input vector and connection weights (the nodal activation) is given by

$$y_k(t+1) = f_k\left(s_k(t)\right) \tag{3}$$

where f_k is the unit's processing function, and the nodal activation $s_k(t)$ is given by

$$s_k(t) = \sum_{l \in U \cup I} w_{kl} z_l(t). \tag{4}$$

For this thesis, the unit's processing function will be either sigmoidal (Eq 1), or a combination of linear and sigmoidal units. Notice in Eq 3 that the output of any unit $y(t+1)$ is not influenced by the external input until time $t+1$. This means that given the current input value at time t, the network will compute (predict) the output for time $t+1$. This fact is important when performing function prediction. In addition, it is important in knowing how to set up the training and testing data sets so that the desired network output is located

at time $t+1$ as opposed to time t.

Since Eqs 3 and 4 specify the entire discrete-time dynamics of the network, the weight update equation must be specified according to these dynamics. This is accomplished by measuring the network performance over time, and then computing its gradient in weight space, following the negative gradient to a minimum total error. For this derivation, the error is defined as

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

where T denotes the set of indices $k \in U$ for which there exists a target value $d_k(t)$ that the output of the kth unit should match. Therefore, the total network error is defined as

$$J_{total}(t) = \sum_t \frac{1}{2} \sum_{k \in U} e_k(t)^2. \tag{6}$$

It is the negative gradient of this total error that must be followed to a minimum value.

The weight update rule adjusts the weight matrix along a positive multiple of the negative gradient of the total error. To achieve this weight update rule, an incremental delta weight (weight change) value is required. This delta weight is initially defined as a fixed multiple of the gradient of the total error with respect to the connection weights at each time step. In other words,

$$\Delta w_{ij}(t) = -\alpha \frac{\partial J(t)}{\partial w_{ij}} \tag{7}$$

where α is some fixed positive learning rate. For this thesis, the learning rate will not be fixed. This modification will be further described in the modification section to follow.

Therefore, following Williams and Zipser's derivation for gradient descent, the algorithm must compute the trajectory by

$$-\frac{\partial J(t)}{\partial w_{ij}} = \sum_{k \in U} e_k(t) \frac{\partial y_k(t)}{\partial w_{ij}} \tag{8}$$

where $\frac{\partial y_k(t)}{\partial w_{ij}}$ is a measure of the sensitivity of the output at time t to a small change in w_{ij}. It is assumed that the initial conditions, external inputs, and remaining weights are not altered at all during this sensitivity measure. Thus the sensitivity of the network at some future time is given by

$$\frac{\partial y_k(t+1)}{\partial w_{ij}} = f'_k[s_k(t)] \left[\sum_{l \in U} w_{kl} p^l_{ij} + \delta_{ik} z_j(t) \right] \tag{9}$$

for all $k \in U$, $i \in U$, and $j \in U \cup I$. The term δ_{ik} denotes the Kronecker delta function. Thus, by defining the variable

$$p^k_{ij}(t+1) = \frac{\partial y_k(t+1)}{\partial w_{ij}},$$

the network dynamics is governed by

$$p^k_{ij}(t+1) = f'_k[s_k(t)] \left[\sum_{l \in U} w_{kl} p^l_{ij} + \delta_{ik} z_j(t) \right] \tag{10}$$

where the initial conditions are defined as

$$p^k_{ij}(t_0) = 0.$$

For this thesis, when the output function is sigmoidal, the derivative of the network processing function with respect to the activation is given by

$$f'_k[s_k(t)] = y_k(t+1)[1 - y_k(t+1)]. \tag{11}$$

When the output function is linear, the derivative of the network processing function with respect to the activation is given by

$$f'_k[s_k(t)] = 1. \tag{12}$$

The RTRL algorithm is a gradient-following algorithm. This means that it follows the gradient descent method for computing weight updates. However, because of it's continuous time nature, it only approximates following the true negative gradient of the error curve. This approximation is done by incrementally updating the weights at each time step rather than by the traditional batch update method, where the weight changes are summed and then added to the existing weights at the end of each epoch. An epoch is simply one complete pass through the entire data set. While the batch method follows the true gradient of the total error, the RTRL technique is known to work well in practice. The use of a small enough learning rate leads to a net weight update whose direction is a close enough approximation to the true gradient.

3.3 Modifications

Several modifications have been added to the RTRL algorithm to better adapt it to the function prediction task. First, the network output was modified to enable the external output units to process as linear units rather than as sigmoidal units. The hidden units remain as sigmoidal processors. Only the external output units were modified. Although the original algorithm did not have linear outputs with sigmoidal hidden nodes, this modification would greatly increase the network's use in applications where the output exceeds the range of 0 to 1. With a linear output, the network's response can be read and interpreted directly. In addition, linear outputs do not require the desired output vector to be normalized, thus saving computation time.

The second modification to the standard RTRL algorithm was to add a variable learning rate to provide for an increased convergence of the total error. The learning rate will be variable based upon the stability of the total error accumulated over an entire epoch. If the ratio of the previous total error to the current total error is less than a desired constant less than one, or if the difference between the previous total error and the current total error is less than zero, then reduce the learning rate by a factor of two (arbitrary). Otherwise, do not change the learning rate. If the difference between the previous total error and the

current total error is less than zero, this means that the total error is beginning to increase or oscillate. By reducing the learning rate at this point, the total error will continue to decrease until convergence. The ratio of the previous total error to the current total error measures the incremental change in the total error in time. If this change is less than the desired incremental change, the learning rate is reduced. The desired incremental change for this thesis is 0.999.

To verify that the learning rate modification increased the network convergence, a series of tests were performed to measure the average total error of the network with and without the variable learning rate. The recurrent network was configured with 1 input, 1 sigmoidal output, and 1 hidden sigmoidal unit. The training data set contained a pseudo-random number sequence 1024 vectors long. Chapter IV contains the results of these tests.

3.4 Testing

The network was tested using several temporally encoded data sets. The first task was to train the network to learn the exclusive OR (XOR) operation. Although XOR is not inherently time dependent, the network will learn it if the output is delayed for a specified amount of time. The next problem attempted was to teach the network to learn an internal state problem (22:97–100). That is, the network must recognize that two particular events have occurred in a prescribed order, regardless of the number of the intervening events. The network was then trained to predict the frequency response of a second order IIR lowpass (Butterworth) filter. IIR is an acronym which means "Infinite Impulse Response". This was done by training the network on the impulse response of the filter, and then testing the response of the network to various inputs.

3.4.1 Exclusive OR (XOR) Exclusive OR (XOR) is a disjoint region problem (see Figure 3). This means that there are two disjoint regions in the decision space for each class. The classes must be separated by at least two decision planes before an input value

can be correctly classified into one of the class regions.

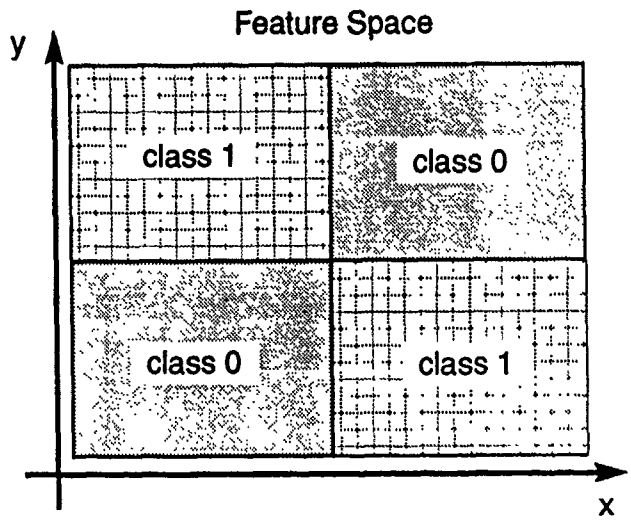


Figure 3. The disjoint region, or exclusive OR, feature space. No single decision plane can separate the regions by class.

Multilayer perceptrons trained with back propagation have demonstrated an ability to learn this problem very well (13:53–61). The use of multiple layers is to allow the formation of multiple decision planes within the feature space.

Since a recurrent neural network can be viewed as a multilayer feedforward neural network which has been folded back onto itself in time (18), it should also be able to solve the XOR problem just as well. However, some alterations to the data set need to be considered. Namely, the desired output of the data needs to be delayed a specified length of time in order to accommodate the predictive nature of the recurrent network. These alterations were required because the XOR problem is not a good test of a recurrent neural network. There is no time dependency within the XOR problem unless it is physically manipulated to contain timed information.

Thus, the XOR problem was included in this thesis in order to identify how the recurrent network performs when given a temporally encoded spatial problem.

The fully connected recurrent network was configured with 2 external inputs, 1 sigmoidal output, and 4 hidden sigmoidal units. The learning rate started at 4.0. Sigmoidal units were chosen because the desired output values are 0 and 1. Initially, the data consisted of a randomly generated set of 1's and 0's as input while the output was the XOR of the input delayed by two time steps. Each data vector was considered a separate time sample. The net trained on a randomly generated binarized XOR data file (that is, the values were either 0 or 1). The number of training vectors was 1024, and the training concluded after 20 epochs. The decision threshold for correctness was 0.5; if the output was greater than 0.5, the output was considered a 1, and if the output was less than 0.5, the output was considered a 0.

After the network training was complete, the weights saved from the training run were used as test weights. The network was tested using another binarized XOR data set generated from a different random seed. This guarantees that the temporal presentation of the XOR data set is randomly changed. The results of this test should show how well the network weights generalized the XOR learning law.

Another separate training was performed to test the network's ability to generalize the complete XOR data set. That is, can the recurrent network learn more that just the verticies of the XOR data set? To answer this question, the network was trained on an analog XOR data set as opposed to the binarized XOR data set. The spatial distrubution of the analog training data set is displayed in Figure 4. The network configuration contained 2 inputs, 1 sigmoidal output, and 5 hidden sigmoidal nodes. The network weights were trained for 300 epochs through the 512 vector-length analog data set. The weights were saved and used to test a 1024 vector-length analog data set. In addition, the two binarized data sets were also tested using the above saved weights. If the network can generalize the analog XOR data set in a true spatial sense, then it would be expected to perform perfectly on the binarized XOR data sets. Chapter IV contains all the results and discussion of these tests.

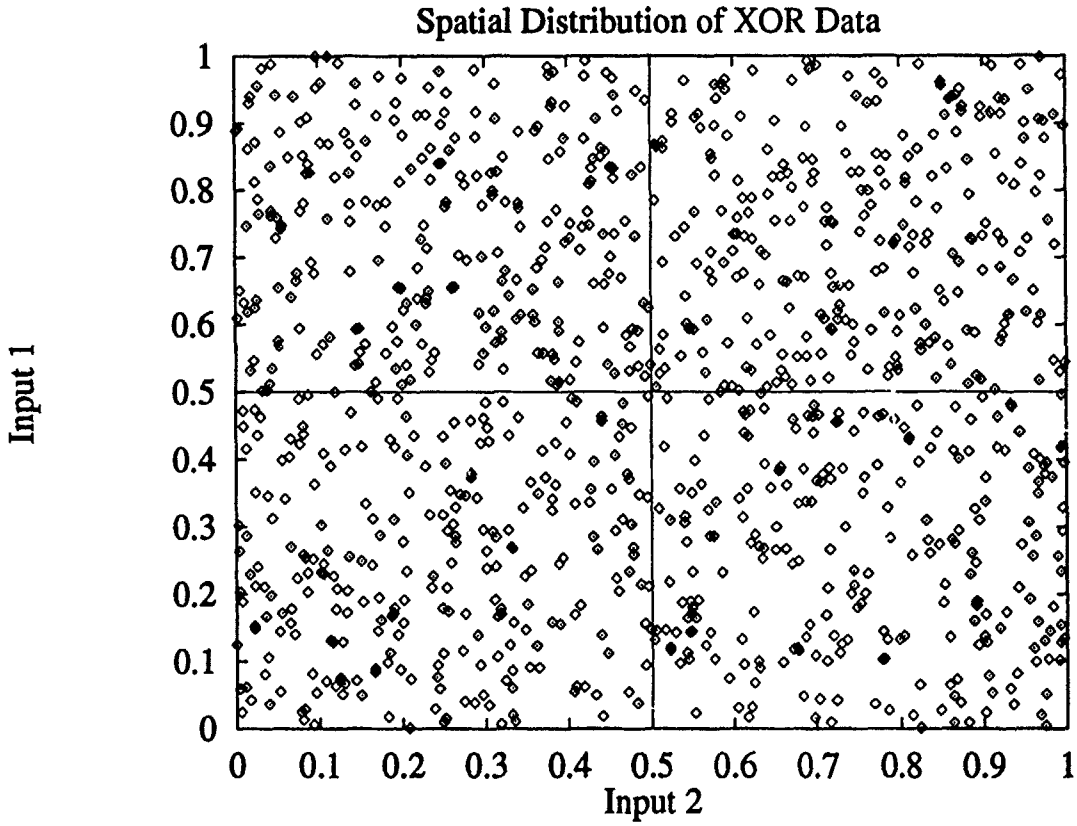


Figure 4. Pseudo-random spatial distribution of the XOR training set used to demonstrate the recurrent network's generalization ability.

3.4.2 Internal State Learning to represent internal state is considered a simple sequential recognition task for humans. However for traditional feedforward neural networks, this is a nearly impossible task. This test should demonstrate the power of a simple recurrent network on timed sequential signals.

As demonstrated by Williams and Zipser (22:97-100), let there be four inputs to the network, each line corresponding to the letters *a b c* and *d* respectively. The *a* and *b* lines are the actual input decision lines and the *c* and *d* lines serve as distractor lines only. On any given time step, a randomly chosen input line is given a value of 1, with all others given the value of 0. The desired output for the network is 1 on the time step immediately following the first occurrence of *b* following an *a*. Otherwise, the desired output is 0.

For this task, the network consists of 4 external inputs, 1 sigmoidal output, 1 sigmoidal hidden node, and an initial learning rate of 5.0. The data set contained 95 time samples (vectors). The initial weight values were randomly generated from the interval [-1,1]. Figure 5 shows the recurrent network configuration used for training on the internal state problem. Because the network algorithm has a variable learning rate, the initial value of the learning rate is used to get the network started on the "right track". For this test, if the learning rate were less than 5.0, the network would still converge but at a slower rate. If the learning rate were greater than 5.0, the network total error would initially converge rapidly but would then rapidly diverge. The algorithm would then lower the learning rate to half of its original value and continue until it converges.

The network was trained and tested on two data sets which contain randomly generated input vectors. However, the time separation between the occurrence of *b* following *a* is different in both data files. This means that the occurrence of the specified state transition differs between the two data sets. For the training set, there were 27 occurrences of *b* while only 10 of these *b*'s followed consecutively after an *a*. The time separation between *ab* pairs for the training set is illustrated in Table 1. For the test set, there were 15 occurrences of *b* while only 8 of these *b*'s followed consecutively after an *a*. The time separation between *ab* pairs for the test set is illustrated in Table 2. Therefore a correct

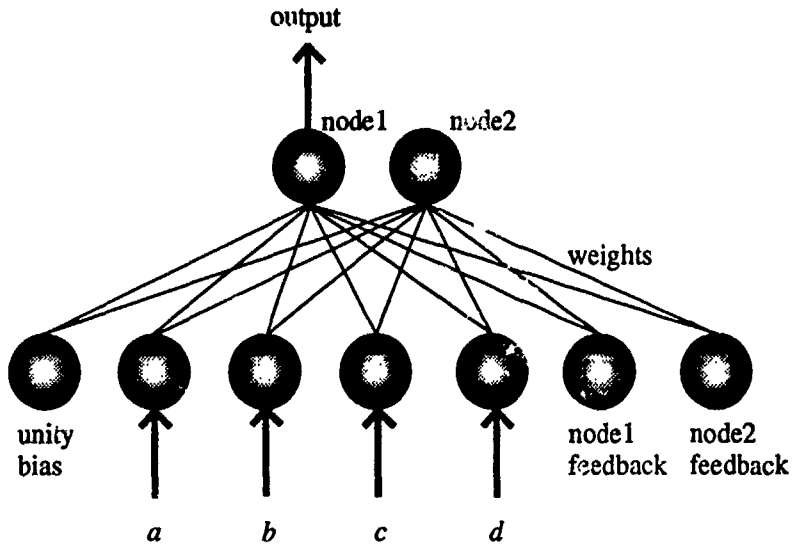


Figure 5. Recurrent network configuration for learning to represent internal state. For this task, the network consists of 4 external inputs, labeled a, b, c, and d, 1 output, 2 sigmoidal nodes, and an initial learning rate of 5.0. The feedback nodes are the previous (t-1) values of both nodes. The entire bottom row of nodes represents the input vector $z(t)$.

prediction of the occurrence of the transition in the test set will demonstrate the network's ability to learn the internal state problem presented. The output of the network and the results of this test are contained in Chapter IV.

3.4.3 Second Order IIR Lowpass Filter This test demonstrates the network's generalization ability in simulating a linear system. A second order lowpass Butterworth filter was used as the linear system for this test. The filter has a normalized cutoff frequency of 0.1 and is described by the following difference equation:

$$y[t] = 0.0676(x[t] + 2x[t-1] + x[t-2]) + 1.1422y[t-1] - 0.4124y[t-2] \quad (13)$$

A data set was generated, using this difference equation, to train the network to learn the frequency response of the filter. The input to the network is the sampled test signal, and the desired output of the network is the output of the difference equation offset by one time

Table 1. The time separation for *ab* pairs in the training data set. The separation is the number of time steps between the occurrence of an *a* and the first occurrence of a *b* in the training set. The occurrence list shows how many of the respective separations exist within the data set.

ab pair separation	Occurrence
1	2
2	3
4	1
5	1
6	1
7	1
18	1
	total = 10

Table 2. The time separation for *ab* pairs in the test data set. The separation is the number of time steps between the occurrence of an *a* and the first occurrence of a *b* in the test set. The occurrence list shows how many of the respective separations exist within the data set.

ab pair separation	Occurrence
1	3
2	1
7	1
9	1
22	1
29	1
	total = 8

step $(t + 1)$. The offset is to accommodate the predictive nature of the recurrent network. Since the input does not affect the output until time $t + 1$, the desired output value in the data set must be located at that $t + 1$ position. The data set contained 128 data points. Appendix C contains the C code (make_data.c) used to generate this data set.

When the input to a system is a single delta function, the output is called the impulse response of the system. Because the Butterworth filter is a linear system, a single delta function (impulse) was used as the input in order to obtain the system's impulse response. A linear system is completely characterized by its impulse response (3:143–144). This means that if the linear system's impulse response is known, the response to a complicated input can be determined by decomposing the complicated input into a superposition of a large number of appropriately weighted and positioned delta functions. The overall response is then determined by summing the responses to all the individual delta functions. Therefore, if the recurrent neural network is trained on the impulse response of the filter, the network's response should completely characterize this linear system, regardless of the complexity of the input signal.

The network consisted of 1 external input, 1 output, and 1 sigmoidal hidden node. For this test, the output node was defined as a linear function. Figure 6 shows the network configuration for the tests.

The network was trained on the impulse response of the filter. This means that the desired output of the network was the actual output of the Butterworth filter when the input was a single impulse. The input to the network was a single impulse. After training, the following input signals were used to test the network's ability to simulate the filter: a unit step, two different sinusoids, and a pseudo-random signal (to simulate white noise). Again, for this test, the network was trained only on the impulse response of the filter.

After the input signal was applied to the network, the output was processed through an FFT (Fast Fourier Transform) algorithm, and the network frequency spectrum was compared to the desired frequency response of the filter. More specifically, the frequency spectrum of the output of the difference equation was compared with the frequency spec-

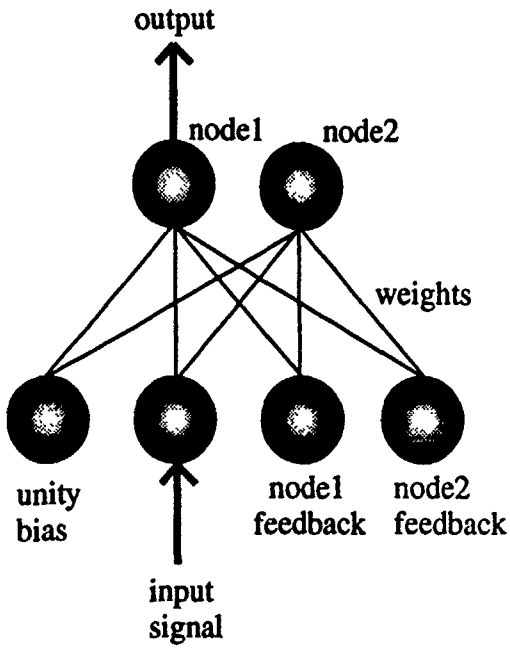


Figure 6. Recurrent network configuration for learning to simulate the frequency response of a linear system. For this task, the network consisted of 1 external input (the input signal to the filter), 1 linear output (the "filtered" input), 1 sigmoidal hidden unit, and a variable learning rate (initially 0.02 for a linear output unit).

trum of the output of the recurrent network. The comparison was performed using all four input signals separately. The results of these tests are highlighted in Chapter IV.

3.5 Applications

If this real-time recurrent learning network can simulate the response of a linear system, the next straightforward application would be to test the predictive ability of the network on real-time problems. Two problems of particular interest are described as follows: predicting 3-D head position in time, and voice data reconstruction.

3.5.1 Predicting 3-D Head Position in Time Given the x, y, and z coordinates of a pilot's head in Euclidean space, the recurrent network should be able to predict what the future position of the pilot's head will be based upon the pilot's previous head position.

For this application, the data set (provided by the Aeronautical Systems Division, Wright-Patterson AFB) contained raw position coordinates as the input vector, and the actual position coordinates for where the head position was in time as the desired output vector. These position vectors were divided into separate coordinate positions. That is, the input and desired output for the x-axis was extracted into a separate data set, and likewise for the y-axis and z-axis data.

The network configuration consisted of 1 input (the current head position at time t), 1 sigmoidal output (the predictive head position at time $t + \tau$, where τ is some arbitrary time), and one sigmoidal hidden unit. The desired output in the data set was offset by 2 time steps. This means that for a given input position, the desired output position is the actual position either 2 time steps in the future. There were 8997 position samples in the data set. The network was trained on the first 1000 samples of the data set for 400 epochs with an initial learning rate of 3.0. Following training, the network weights were used to test the remaining 7997 data points to see how well the recurrent network could predict the respective coordinate position.

The training results are then compared to the results of a statistical prediction algorithm which theoretically produces the best linear approximation. The statistical prediction algorithm is fully described in Appendix E. This comparison is expected to show how the recurrent network performs with respect to the best linear prediction. Chapter IV contains all the results of this application of the recurrent network.

3.5.2 Voice Data Reconstruction For the task of voice data reconstruction, the recurrent network was required to learn the difference between fricative (noisy speech) and voiced (non-noisy) speech samples. More precisely, given a select set of input features which should uniquely describe a sampled portion of speech, the recurrent network was to classify whether that portion of speech was fricative (class 1) or voiced (class 0). If the sampled portion of speech was classified as a fricative, noise was added to that portion of the signal on reconstruction, whereas, if the sampled portion was classified as voiced, no

noise was added during reconstruction. Thus the reconstructed speech signal would regain most of the high frequency content it lost before transmission (20).

The data set for this application consisted of four features computed from a variable width sample of speech. The first feature was the total energy contained in the sample. The second feature was the number of zero-crossings that occurred during the sample period divided by the sample window length. The third feature was the number of slope changes divided by the sample window length. And the fourth feature was the total energy below 500 hertz within the sample window. The desired output was a classification based upon whether the input features described a noisy portion of speech or not.

Previous attempts at learning the data set used a feedforward neural network trained with backpropagation. This network contained 4 inputs, 4 hidden sigmoidal nodes, and a 2 class output. After training, the network weights were saved and used as previously described in the reconstruction process to classify noisy or non-noisy data. Training accuracy was 87% . However, this feedforward network would not make the proper decisions regarding the noise classification of the data when used in the reconstruction program. Therefore, another network architecture was sought out to attempt to solve the problem: the recurrent network.

The recurrent network configuration consisted of 4 inputs, 1 sigmoidal output, and no hidden units. The network was trained for 400 epochs on a data set containing 1500 data samples. The learning rate was initialized to 0.02. If the learning rate were greater than 0.02, the total error for a single epoch would be exceedingly large, causing the network to catastrophically diverge.

Following training, the network weights were saved and used in the reconstruction program. Within the reconstruction program, the weights were used with the transmitted signal to reconstruct the speech pattern, adding noise where needed based upon the network's classification decision. Chapter IV contains all the results of this application of the recurrent network.

3.6 Summary

The methodology for developing and testing the RTRL algorithm has been described. The dynamics of the RTRL algorithm were described and the modifications to the original algorithm were outlined. Next, the testing methodology used in this thesis was described. The results of these tests show how robust the recurrent network is to learning temporal XOR, representing internal state, and simulating a linear system. The predictive ability of the recurrent network was then applied to two problems: head position tracking, and voice data reconstruction. Chapter IV contains the results and a discussion of these tests.

IV. Results and Discussion

Chapter III covered details of the development, modification, and testing of the RTRL algorithm as a viable part of a function prediction scheme. This chapter contains the results of these tests, including a section on the recurrent network's application to two function prediction problems: 3-d head position tracking, and voice data reconstruction. The results are presented in the same order as they appeared in Chapter III.

4.1 Modifications

To verify that the learning rate modification increased the network convergence, a series of tests were performed to measure the average total error of the network with and without the variable learning rate. The recurrent network was configured with 1 input, 1 sigmoidal output, and 1 hidden sigmoidal unit. The training data set contained a pseudo-random number sequence 1024 vectors long. The recurrent network was trained on this data set for 500 epochs. The results displayed do not indicate that the network learned to predict the pseudo-random sequence. Rather, the plots simply illustrate the difference between using the variable learning rate modification and a fixed learning rate.

Figure 7a) displays the network training error when the learning rate is fixed at 4.0. The total error decreases to a point and then diverges and becomes unstable. This implies that the learning rate is not small enough to account for small weight variations required within the network. Figure 7b) shows the exact same training as before, except with a variable learning rate, initially set at 4.0. The network continued to converge throughout the 500 epochs. The sharp drops in the error plot show when a change in the learning rate occurred. The final learning rate value at the end of the training was 0.5. Using this final value as the starting value for a fixed learning rate test, the network was trained again on the same data set. Figure 7c) displays the results of this last test. Note that the network continues to smoothly converge throughout the training run. Yet at the last epoch, the total error was still not as low as that for the variable learning rate. This shows that a

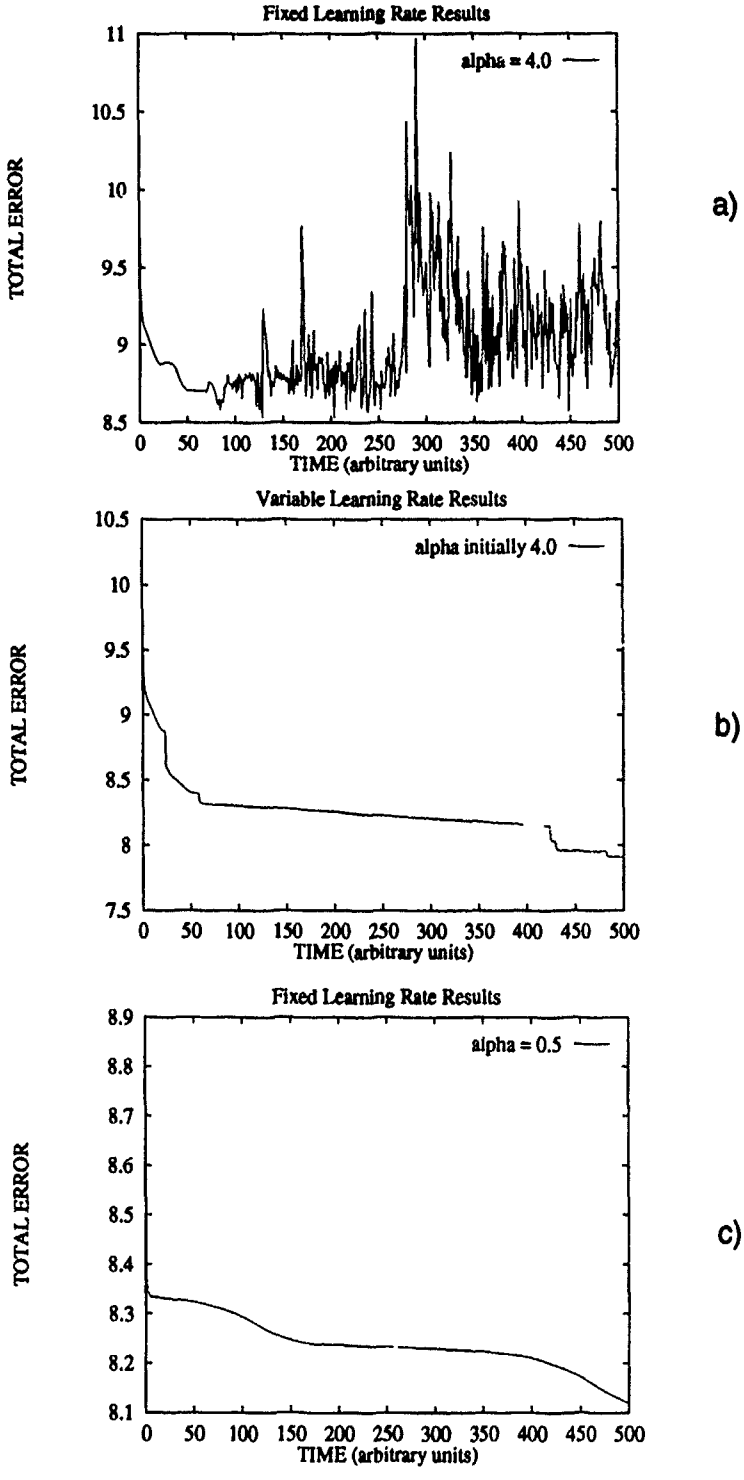


Figure 7. Results of using a variable learning rate. The total error versus epoch number shows training using a) a fixed learning rate, b) a variable learning rate, and c) a lower fixed learning rate.

large learinig rate initially, followed by a successively decreasing learning rate converges rapidly when possible, and slowly when necessary. Therefore, the variable learning rate modification has shown to be successful in increasing the network convergence to an overall lower total error.

4.2 Exclusive OR

The first problem used to test the recurrent networks ability to learn was the classic XOR problem. The recurrent network was initially trained on the binarized data set described in the Chapter III. The learning accuracy for this binarized training data set was 100% after 20 epochs, with a total squared error on the final epoch of 0.032. The decision threshold for correctness was 0.5; if the output was greater than 0.5, the output was considered a 1, and if the output was less than 0.5, the output was considered a 0. However, this accuracy was meaningless unless the networks generalization ability was tested on a separate data set.

For this test, the weights saved from the training run were used to test a separate binarized data set generated using a different randomization seed. During the test, the weights remained fixed, and the network processes the test data only once. In this test run, there were zero prediction errors. Thus, the generalization accuracy of the network was 100% when tested on the separate binarized test set. In fact, there were zero errors for four other binarized test sets, all of which were generated from separate seeds.

However, a separate test was performed to see how the network learns when trained on an analog data set. After 300 training epochs (512 input vectors) on the analog data set, the training accuracy was 98.1% correct, based on a decision threshold equal to 0.5. The network weights were saved and used to test a 1024 vector-length test set containing randomly generated analog values. After testing, the prediction accuracy was 93.1% correct. Then two binarized data sets were tested using the weights computed from the training run on the analog XOR data set. The results of the binarized data tests were as follows: for the first binarized data set, the testing accuracy was 91.3% correct, and for

the second binarized data set, the testing accuracy was 90.8% correct.

These results were not really expected. It was expected that the network trained on the analog XOR data set would be able to exactly learn the corner values (vertices). However, the above results show that almost 10% of the data points in the binarized test set were incorrectly classified. This implies that the network did not learn the true spatial XOR problem. Rather, it learned the spatio-temporal XOR problem. In other words, the network learned the spatial XOR problem as presented in a temporal sequence.

To further demonstrate that the network did not learn the true XOR problem, Figure 8 contains a spatial distribution plot of the analog XOR data set which identifies the points which were classified correctly or incorrectly following testing. The diamonds represent the points within XOR space which the network incorrectly classified. Notice how evenly distributed the misclassified points are throughout the XOR regions. There were bad decisions made in every region, even points very close to the vertices (corners). If the recurrent network truly learned the spatial XOR problem, the bad decisions would be expected to lie close to the intersecting lines (cross-hairs) between the respective XOR regions.

Two more analog XOR data sets were created to see if the spatial decision regions change as the temporal presentation of the analog XOR data changes. These new analog XOR sets are identical except that the input sequence of XOR data is changed. The first set will be referred to as test case 1, and the other as test case 2. Using the same weights generated from training on the previously mentioned analog training set, the testing accuracy for test case 1 was 90.1% correct, and for test case 2 was 90.5% correct. Figure 9a) displays the spatial dicisions for this test case 1 and Figure 9b) displays the spatial dicisions for this test case 2. This time, notice the difference in spatial location of the misclassified points in Figure 9a)to the spatial location of the misclassified points in Figure 9b). Since the temporal presentation of the data set changed, the decisions made by the network changed also.

Therefore, it is suggested that the actual decision region the recurrent network uses

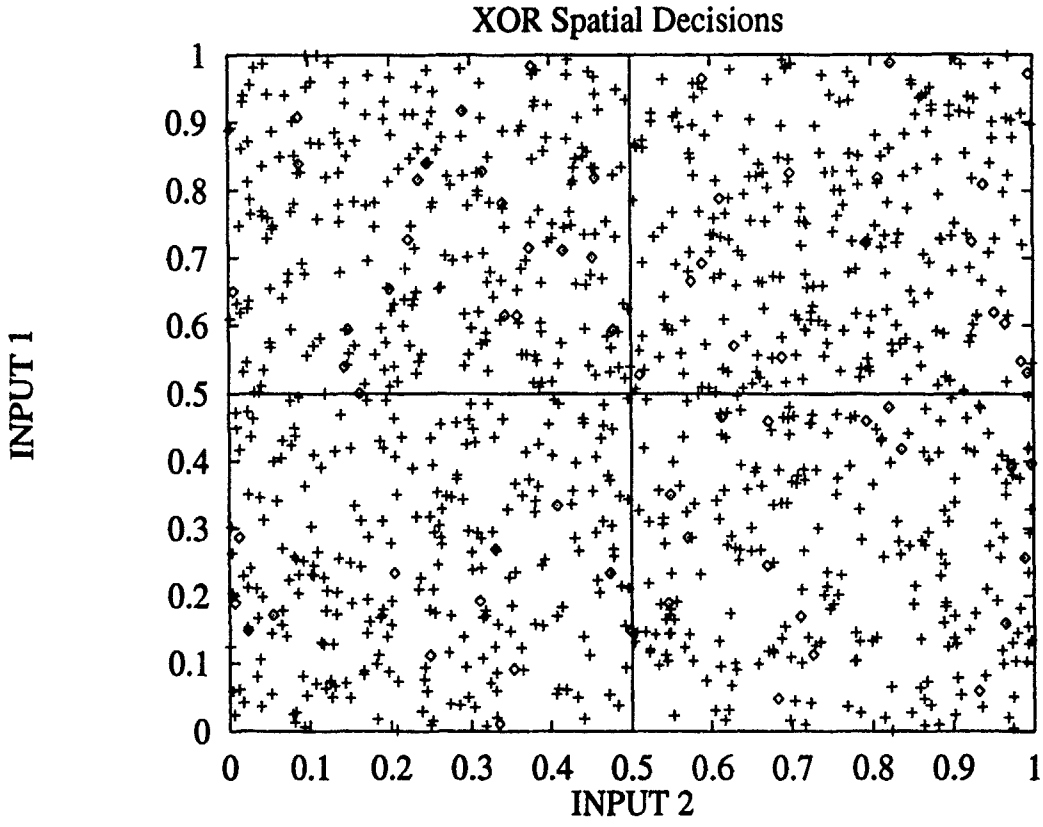
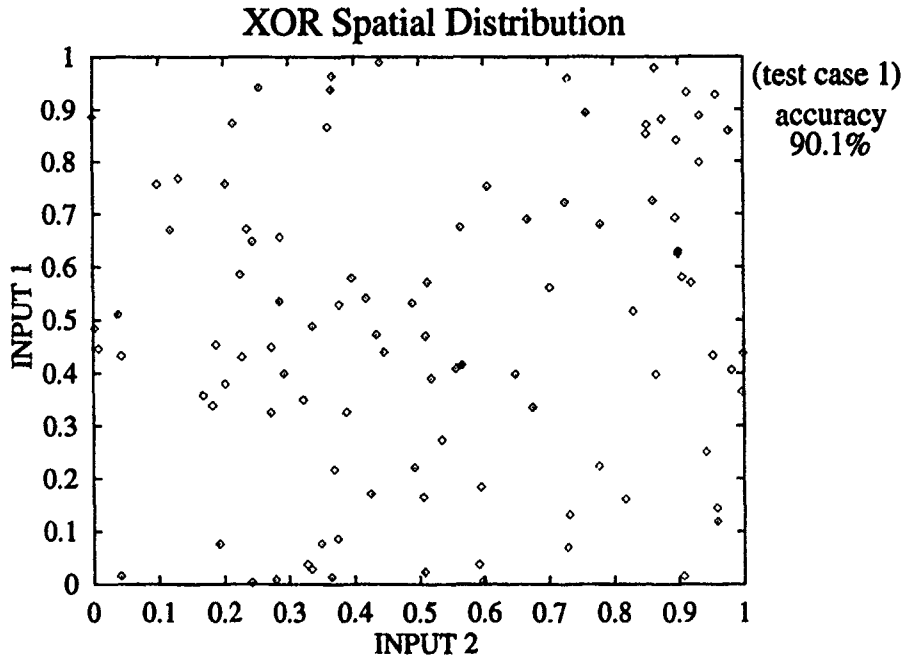
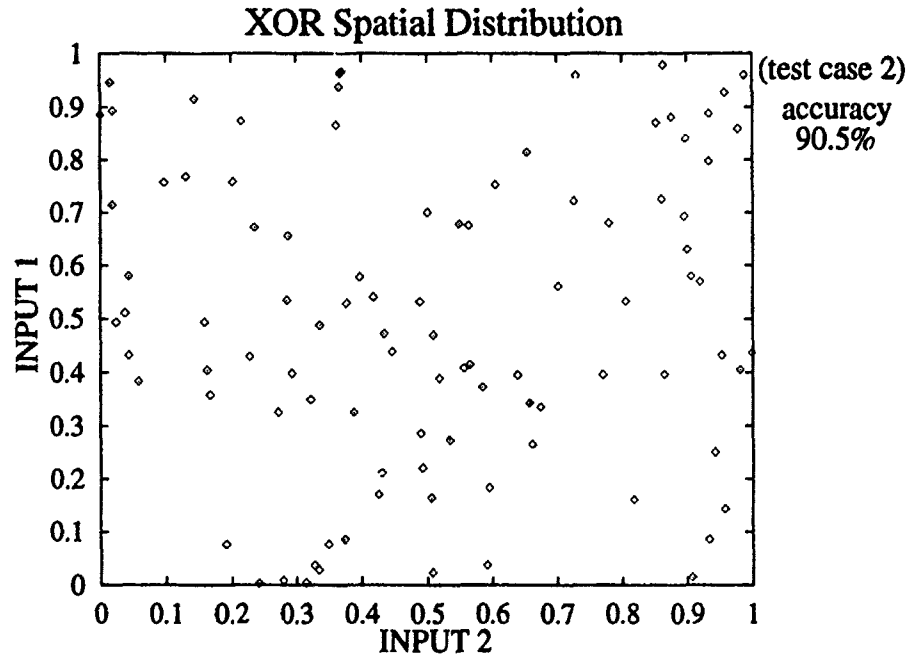


Figure 8. XOR spatial distribution decision plot. The diamonds represent the points in XOR space that the network incorrectly classified. There were bad decisions made in every region, even for points very close to the vertices (corners). No clear spatial decision region can be found. All of the incorrect decisions were expected to be located near the cross-hair lines.



a)



b)

Figure 9. (a) XOR spatial distribution decision plot for test case 1. (b) XOR spatial distribution decision plot for test case 2. The diamonds represent the points in XOR space that the network incorrectly classified.

is described by a nonlinear spatio-temporal mapping. Previous research suggests that the recurrent network learns the XOR problem by organizing itself into an appropriately layered feedforward network (22:96–97). However, if this were truly the case, the recurrent network would simply create a clearly discernable spatial decision region as was previously expected. But as the previous results have shown, no clear spatial decision region was formed. The recurrent network's XOR decision was made based on the temporal presentation of the spatial information contained in the data set.

4.3 Internal State

The network's ability to represent internal state is outlined in the following plots. In Figure 10, the output of the network is compared to the desired training output in time. As outlined in Chapter III, the training data set consisted of a randomly generated input line (a, b, c, or d), one of which set equal to 1 and the rest set equal to 0. The desired output is 1 on the time step immediately following the first occurrence of a b following a, and 0 otherwise. Figure 10 shows how the trained network was able to predict the desired output after only 20 epochs through a 95 vector data set. These weights were saved and used to test the network's generalization ability on a separate data set.

Figure 11 details how well the network can generalize from the training data set to an application on a specific test data set. These results show a 100% accuracy in predicting the occurrence of a state transition of as much as 29 time steps apart. Reference Tables 1 and 2 in Chapter III for the occurrence of ab pair and the respective time separations. These results imply a specific capacity to remember. Yet these results also imply that the network will identify the first occurrence of a b line transition no matter how long ago the first a transition occurred.

In analogy, the network configures to be a set-reset state device (flip-flop). Reference Table 3 for the following discussion. The occurrence of an a sets the state of the hidden unit high while the occurrence of a b during a low state resets the state of the hidden unit to low. If the previous state was high, the output will be high. But if the previous state was

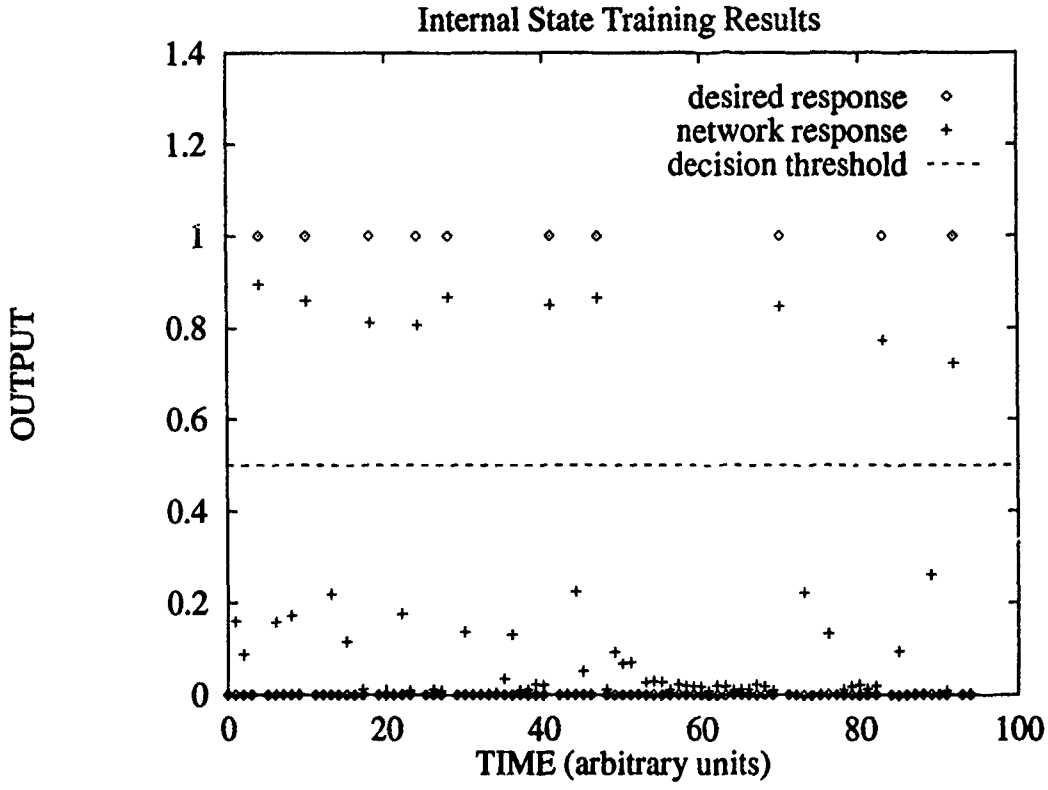


Figure 10. Desired and actual output of the recurrent network after training. The decision threshold for correctness was 0.5; if the output was greater than 0.5, the output was a 1, and if the output was less than 0.5, the output was 0. The separation of outputs greater than 0.5 represents the separation time between when an *a* occurred and the first *b* occurred. Training accuracy was 100%.

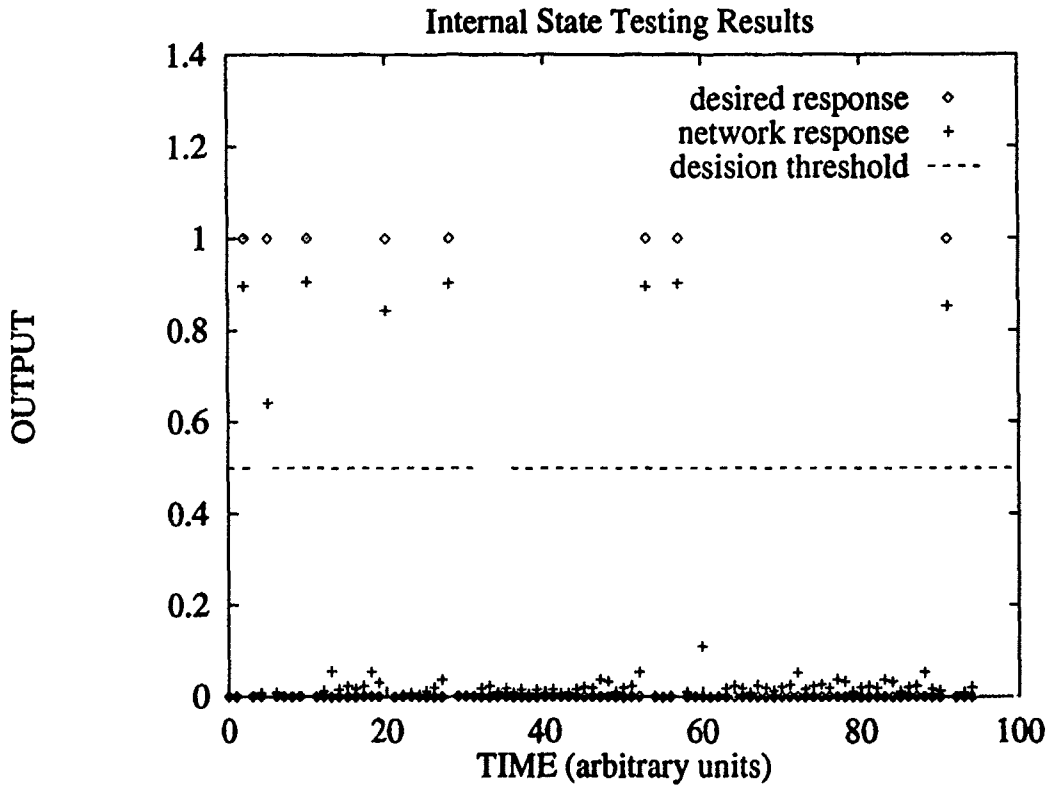


Figure 11. Internal state test output of the recurrent network after training 20 epochs. The decision threshold for correctness was 0.5. The separation of outputs greater than 0.5 represents the separation time between when an *a* occurred and the first *b* occurred. Testing accuracy was 100%.

low and a b occurs, the output will remain low because the hidden unit has not been set low. When the previous output was high, the recurrent weights for node one will always reduce this high to a low, thus resetting the output until the next state transition occurs.

Table 3. Training weights for the internal state problem. The first row are weights for the output node and the second row are weights for the hidden node.

	bias wt	a wt	b wt	c wt	d wt	recur wt 1	recur wt 2
output node	-6.636	-1.656	3.749	-2.582	-2.209	-2.877	5.480
hidden node	-1.565	4.242	-4.055	-1.535	-0.609	-3.461	5.860

4.4 Second-Order IIR Lowpass Filter Simulation

As described in Chapter III, the recurrent network was trained to simulate second-order IIR lowpass filter (Butterworth). The following input signals were used to test the network's ability to accurately simulate the filter's response: a unit step, a cosine wave, an inverted sine wave, and a pseudo-random number sequence (to simulate white noise). Figure 12 shows the desired frequency response of the Butterworth filter.

4.4.1 Impulse Response The network was trained on the impulse response of the filter. This was accomplished by generating a data set using the difference equation displayed in Eq 13. The input to the generator was an impulse $\delta(t)$, where $\delta(t)$ equals 1 for $t = 0$ and $\delta(t)$ equals 0 otherwise. The output of the generator was used as the desired output of the network. The desired output of the network was simply the output of the generator delayed by one time step. In theory, the impulse response of a linear system completely describes the system. Therefore, by training the network on the impulse response of the system, the network should be expected to accurately simulate the response of a linear system to any other input.

Figure 13 contains the results of the network after training for 600 epochs on the desired impulse response of the filter. The output of the network was processed through a

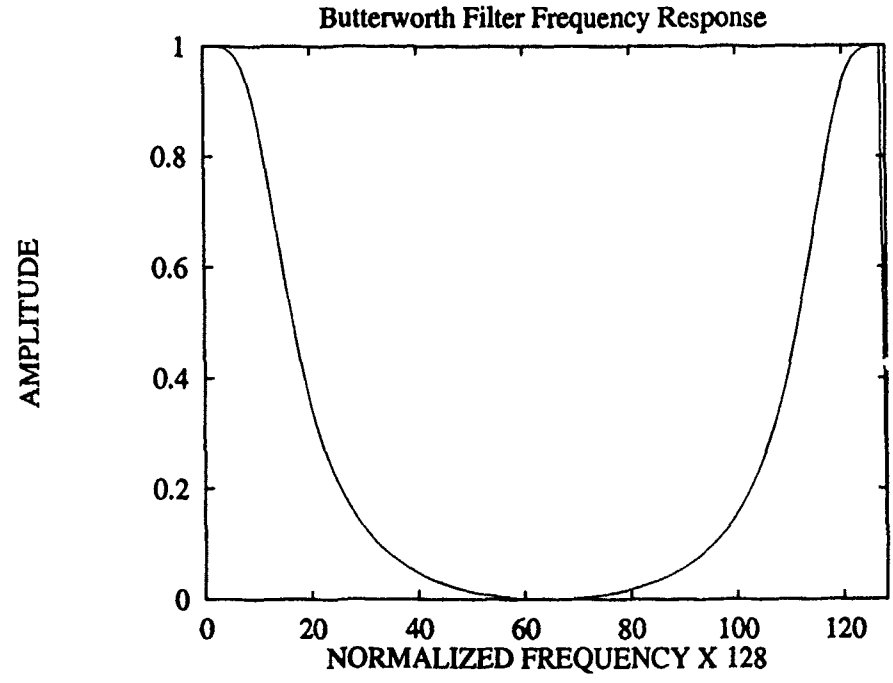


Figure 12. Desired frequency response of the Butterworth filter.

fast Fourier transform (FFT) algorithm and the results plotted in Figure 14. The weights generated by the network after this training run were saved and used to test the network's response to other input signals.

Based on the results in Figure 14, the recurrent network could not completely memorize the impulse response of the filter, and thus, a complete filtering of higher frequency components could not be learned. This is shown by the non-zero response in the region where no frequency components should be. However, the amplitude of the higher order frequency components will still be greatly attenuated. This does not imply that the network did not learn to generalize the response of the Butterworth filter. The real test is to apply the weights generated from this training to other input signals and compare the results to the expected filter response.

4.4.2 Unit Step Response Using the weights generated by the network after it was trained on the Butterworth filter's impulse response, the network was tested using a unit

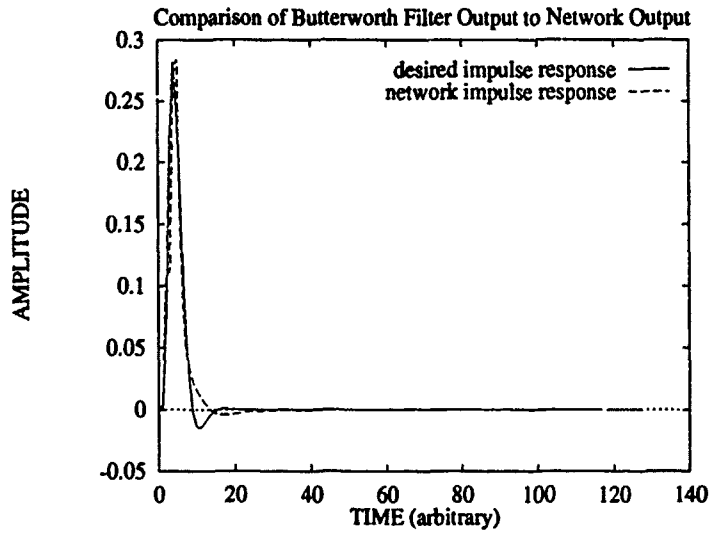


Figure 13. Comparison of the network's impulse response to that of the desired impulse response of the Butterworth filter after 600 training epochs. The impulse response completely characterizes the filter.

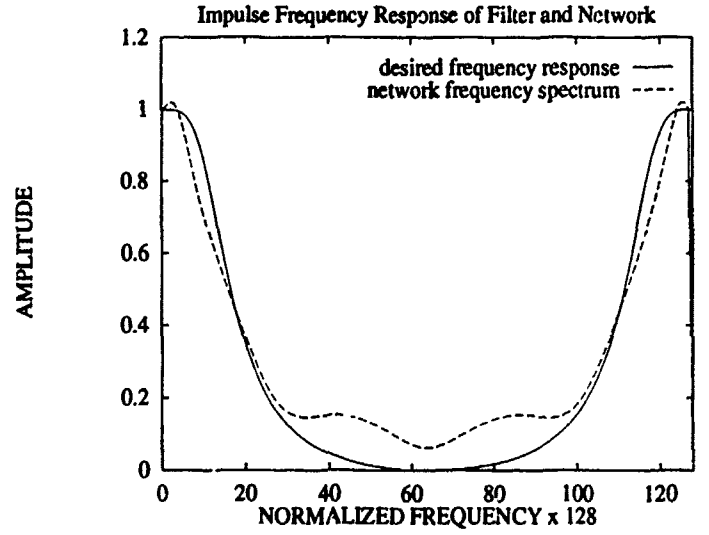


Figure 14. Comparison of the network's impulse frequency spectrum to that of the desired frequency response of the Butterworth filter after 600 training epochs. Since the recurrent network could not completely memorize the impulse response of the filter, a complete filtering of higher frequency components could not be learned. However, the amplitude of the higher order frequency components will be greatly attenuated.

step as the input signal. A unit step is defined as equal to 1 for $t \geq 0$ and equal to 0 otherwise. The network response to the step input is shown in Figure 15, and Figure 16 shows the frequency domain representation to the same step input.

The results plotted in Figure 15 show how well the recurrent network's response matched the expected response of the Butterworth filter. The one big difference is the lack of an overshoot in the network's response. This is a feature common to a heavily overdamped system, where as the Butterworth filter's response only shows slight damping. Nevertheless, the network still showed an excellent ability to simulate the step response of the filter. In the frequency domain results shown in Figure 16, small differences can be noted throughout the plot. However, since the plot is log-linear, these differences are amplified.

4.4.3 Sinusoidal Response The network's response was further tested by using two different sinusoidal waves as inputs to the system. The first sinusoid was a cosine wave that completes 2 cycles within 128 sample points. The response of the trained network to this cosine wave is shown in Figure 17. Throughout the plot of the response, the network response very closely predicted the expected response of the Butterworth filter.

In the frequency domain, it is apparent that this cosine wave was completely within the passband of the filter. Figure 18 displays the actual network spectral response compared to the expected Butterworth spectral response. As with the unit step response, the cosine frequency response of the network so closely matched the cosine frequency response of the filter that no significant differences can be noted.

The second sinusoid used to test the recurrent network's ability to simulate the response of the Butterworth filter was an inverted sine wave that completes 4 cycles within 128 sample points. Figure 19 displays the response of the trained network compared with the expected Butterworth filter response. It illustrates how closely the network response predicted the expected response of the Butterworth filter. Figure 20 simply shows how well the network learned the response of the Butterworth filter.

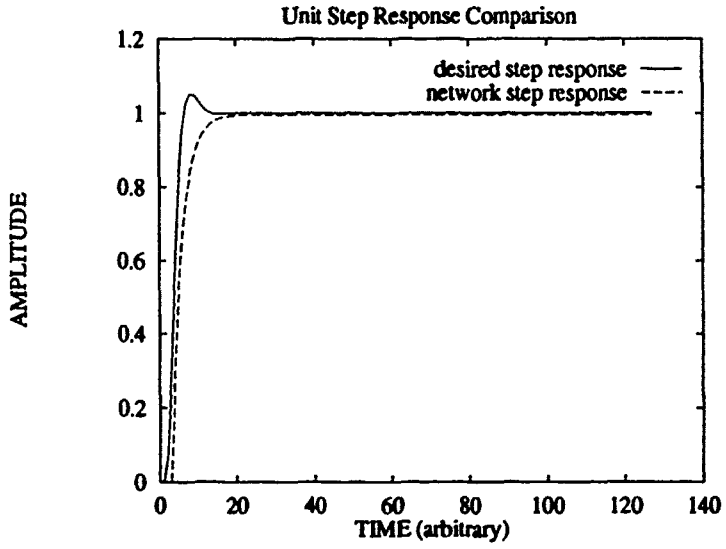


Figure 15. Comparison of the recurrent network's response to a unit step input with the Butterworth filter's response to a unit step input.

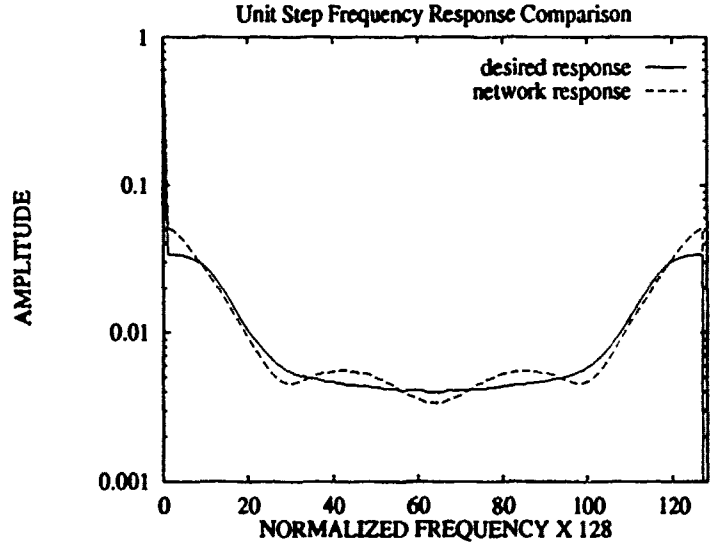


Figure 16. Comparison of the recurrent network's spectral response to a unit step input with the Butterworth filter's spectral response to a unit step input. The plot is log-linear.

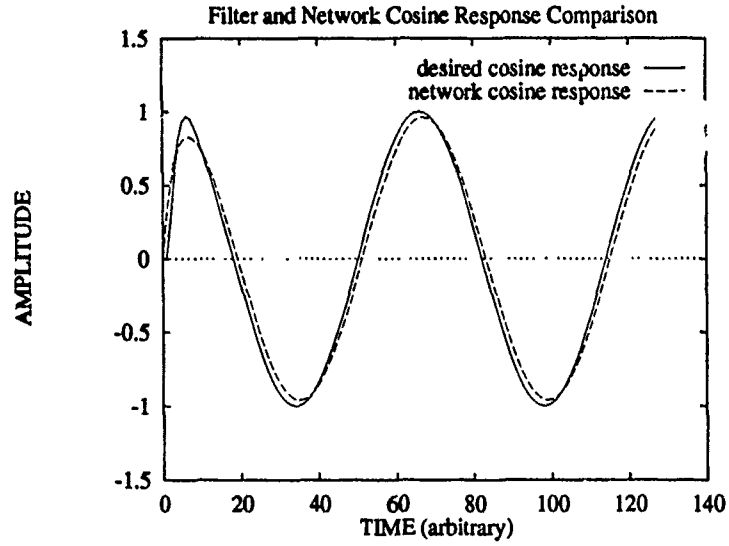


Figure 17. Comparison of the recurrent network's response to a cosine wave input with the Butterworth filter's response to a cosine wave input.

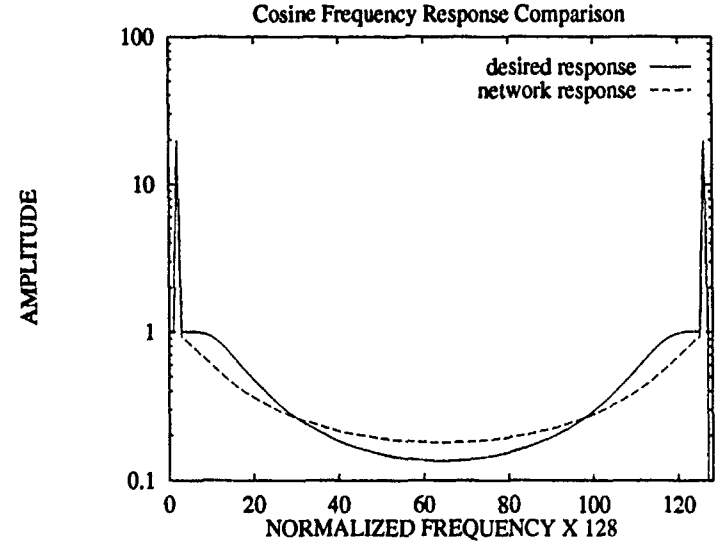


Figure 18. Comparison of the recurrent network's spectral response to a cosine wave input with the Butterworth filter's spectral response to a cosine wave input. The plot is log-linear.

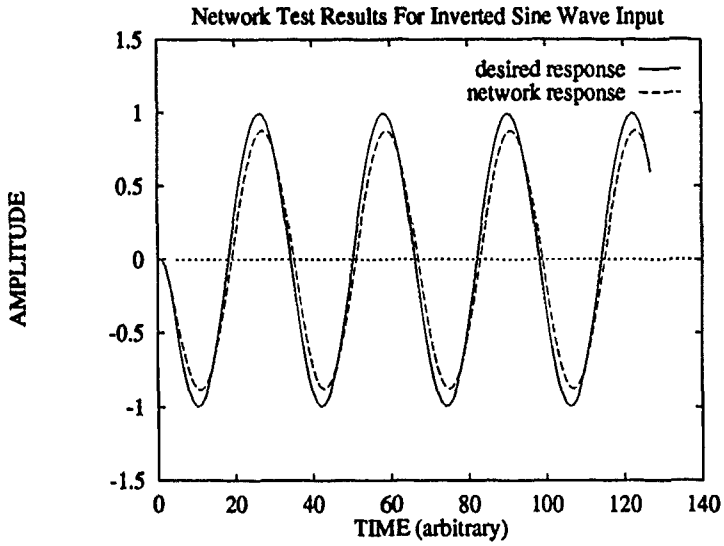


Figure 19. Comparison of the recurrent network's response to an inverted sine wave input with the Butterworth filter's response to an inverted sine wave input.

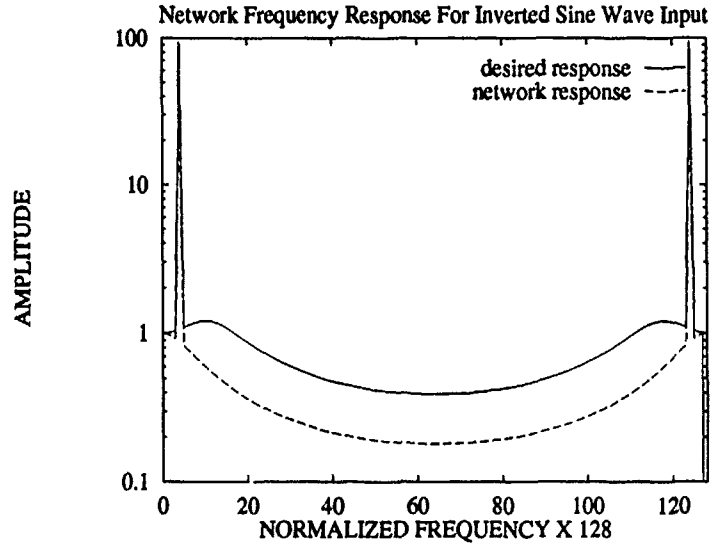


Figure 20. Comparison of the recurrent network's frequency response to an inverted sine wave input with the Butterworth filter's frequency response to an inverted sine wave input.

4.4.4 Pseudo-Random Number Sequence Response The last test of the recurrent network's ability to simulate the response of a Butterworth filter was to apply a broadband, noisy signal to the input of the network. The noisy signal was approximated by a pseudo-random number sequence in the range [-1,1]. Figure 21 shows the results of the network's response to a noisy input signal compared to the expected response of the Butterworth filter to the same noisy signal. Although the network response does not exactly follow the expected response, it does follow the expected response close enough to say that the network has indeed learned to simulate the response of the Butterworth filter for a noisy input signal.

In the frequency domain plot displayed in Figure 22, it is clear to see how the frequency components in the cutoff region of the filter have been attenuated when compared to the amplitude of the frequency components in the filter's passband. As identified earlier in the training of the impulse response, the recurrent network did not completely memorize the impulse response of the Butterworth filter. Thus, those frequency components falling outside the filter's cutoff region will only be attenuated and not completely cutoff.

4.5 Predicting 3-D Head Position in Time

The recurrent network configuration consisted of 1 input (the current head position at time t), 1 sigmoidal output (the predictive head position at time $t + \tau$, where τ is some arbitrary time based on the sampling rate of the system), and 1 sigmoidal hidden unit. The desired output in the data set was offset by $\tau = 2$ time steps. This means that for a given input position, the desired output position is the actual position displaced 2 time steps in the future. There were 8997 position samples in the data set. The network was trained on the first 1000 data points for 400 epochs with an initial learning rate of 3.0. Figure 23 illustrates how close the recurrent network predicted the head's y-position for $\tau = 2$. Only the y-position was displayed because the x- and z-position plots were both equally as accurate as the y-position plot.

Notice how the network prediction silghtly lags behind the actual y-position. This

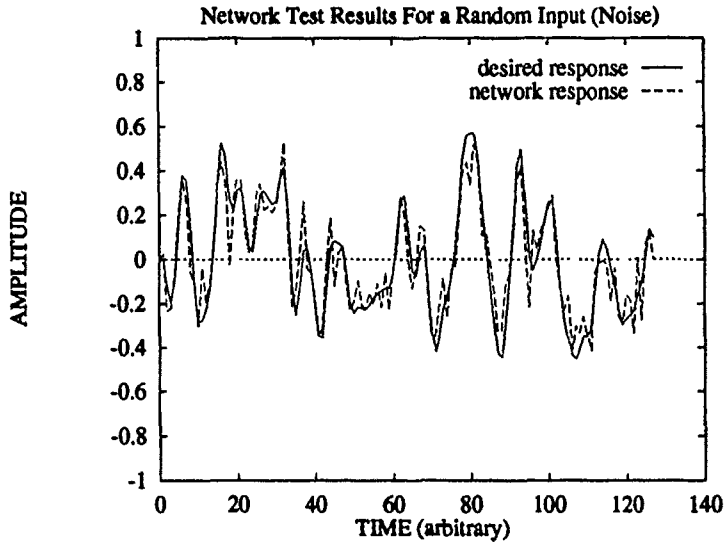


Figure 21. The results of the recurrent network's response to a noisy input signal compared to the expected response of the Butterworth filter to the same noisy signal.

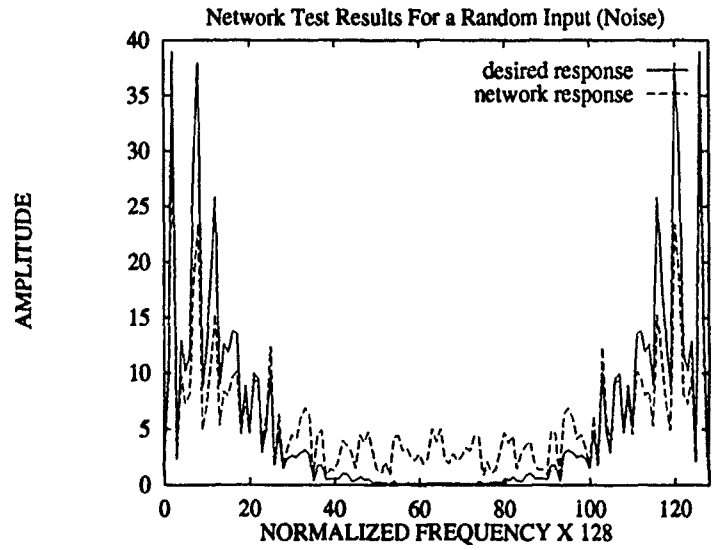


Figure 22. The results of the recurrent network's frequency response to a noisy input signal compared to the expected spectral response of the Butterworth filter to the same noisy signal.

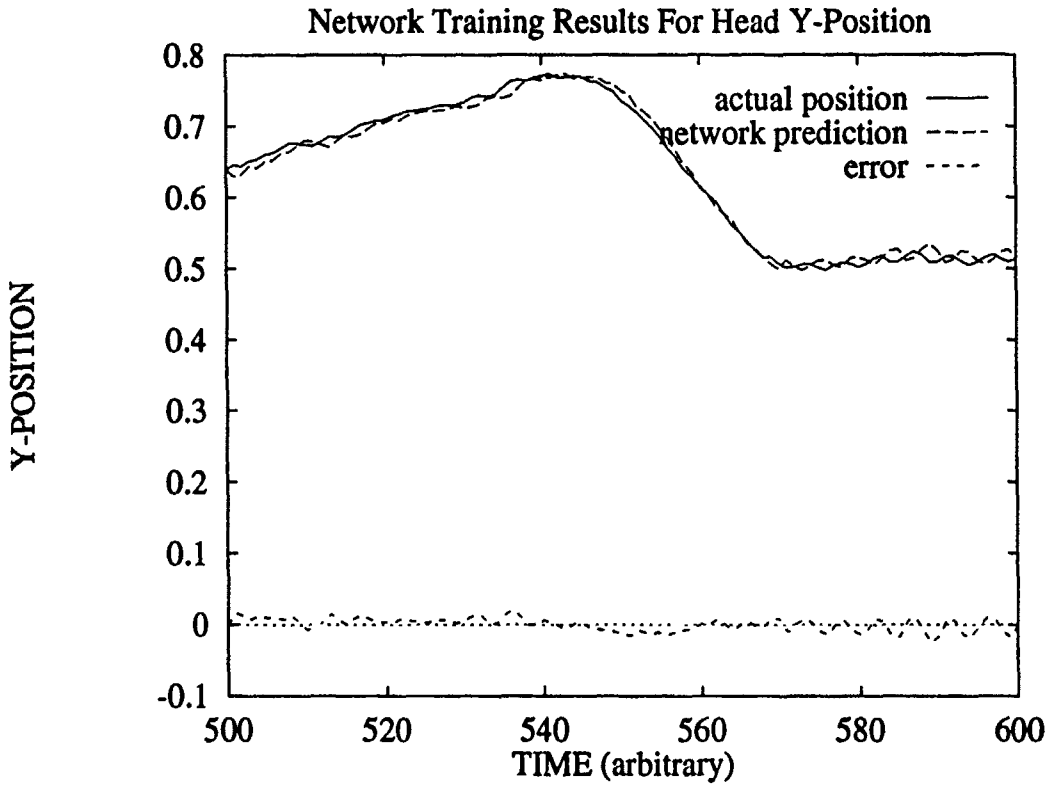


Figure 23. Predicting head position training results. The recurrent network trained for 400 epochs on 1000 data points. Only a portion of the results are displayed. The network output was trained to predict the value of the input function two time steps in the future. This plot is a comparison of the network output $y(t+2)$ to the actual y-position at time $t+2$.

indicates that the network did not learn to accurately predict the pilot's head position in time. Portions of the networks prediction are very close to the actual values, but this is not consistant throughout the data set.

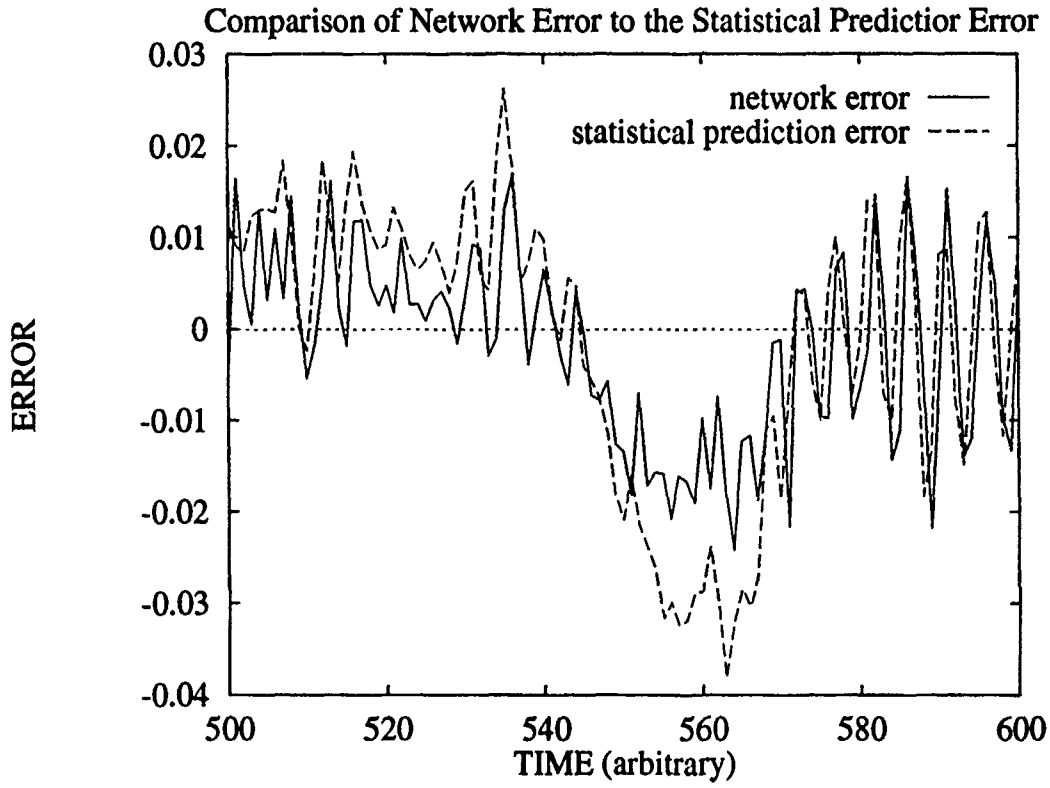


Figure 24. This plot is a comparison of the recurrent network's error $e(t)$ to the statistical prediction error.

So, how well does the network compare to the best linear prediction? A statistical prediction algorithm was used for this comparison. The same data set used to train the network was used in the statistical prediction algorithm. The results of this comparison (shown in Figure 24) show that the network only slightly outperforms the statistical predictor. The total mean squared error of the network was 0.000174 while the total mean squared error of the statistical predictor was 0.000220. However, in terms of prediction, portions of the network's output were very close to the actual signal whereas

the statistical prediction consistantly lagged behind the actual signal. On this point, the network performance was better.

Thus, from these results, the recurrent network shows to be a more robust function predictor than the best linear predictor. This is true for three reasons. One, the network does not require that the entire temporal sequence be known while the statistical predictor does. Two, the network does accurately predict portions of the pilot's head position in time while the statistical predictor always lags. Three, the network can be trained in real-time and updated as necessary to accommodate unexpected future events whereas the statistical predictor can not.

4.6 Voice Data Reconstruction

For the task of voice data reconstruction, the recurrent network was required to learn the difference between a fricative (noisy) and a voiced (non-noisy) portion of speech. The recurrent network configuration consisted of 4 inputs, 1 sigmoidal output, and no hidden units. The network was trained for 400 epochs on a 1500 vector-length data set. The learning rate was initialized to 5.0. The data set for this application consisted of four features computed from a variable width sample of speech. The first feature was the total energy contained in the sample. The second feature was the number of zero-crossings that occurred during the sample period divided by the sample window length. The third feature was the number of slope changes divided by the sample window length. And the fourth feature was the total energy below 500 hertz within the sample window. The desired output was a classification based upon whether the input features described a fricative (class 1) or voiced (class 0) portion of speech. The classification was assessed purely on human discretion.

Figure 25 displays the results of the recurrent network after training for 400 epochs through the 1500 vector-length data set. The decision accuracy of 98.4% was based upon whether the network output was greater than 0.5 for a class 1 or less than 0.5 for a class 0. The few classification errors the network made were for noisy regions that contain higher

than normal energy (such as the sound for the letter "k").

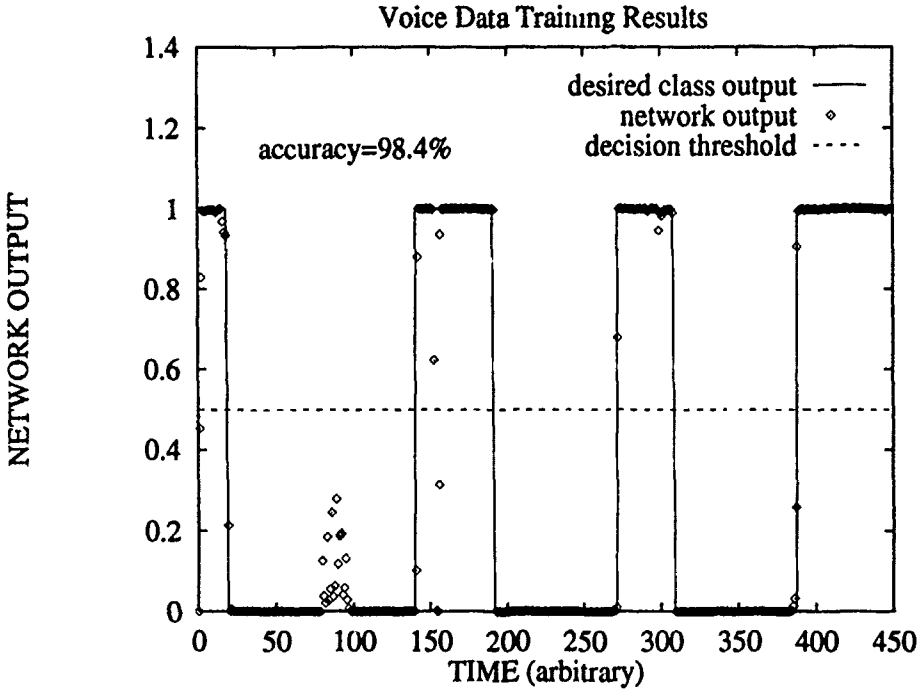


Figure 25. Voice data classification results after training on 400 epochs. The few classification errors the network made are for noisy regions that contain higher than normal energy (such as the sound for the letter "k").

Following training, the network weights were saved and used in the reconstruction program. Within the reconstruction program, the weights were used with the transmitted signal to reconstruct the speech pattern, adding noise where needed based upon the network's classification decision. Figure 26 illustrates how the network weights were used in the reconstruction program to classify a given speech pattern. The fricative regions classified as "1" were noisy regions where noise was added to the signal during reconstruction. In the voiced "0" regions, no noise was added to the signal during reconstruction. Using this network, the reconstruction program was able to reproduce an intelligible voice signal whereas the decisions made by the feedforward network previously used could not.

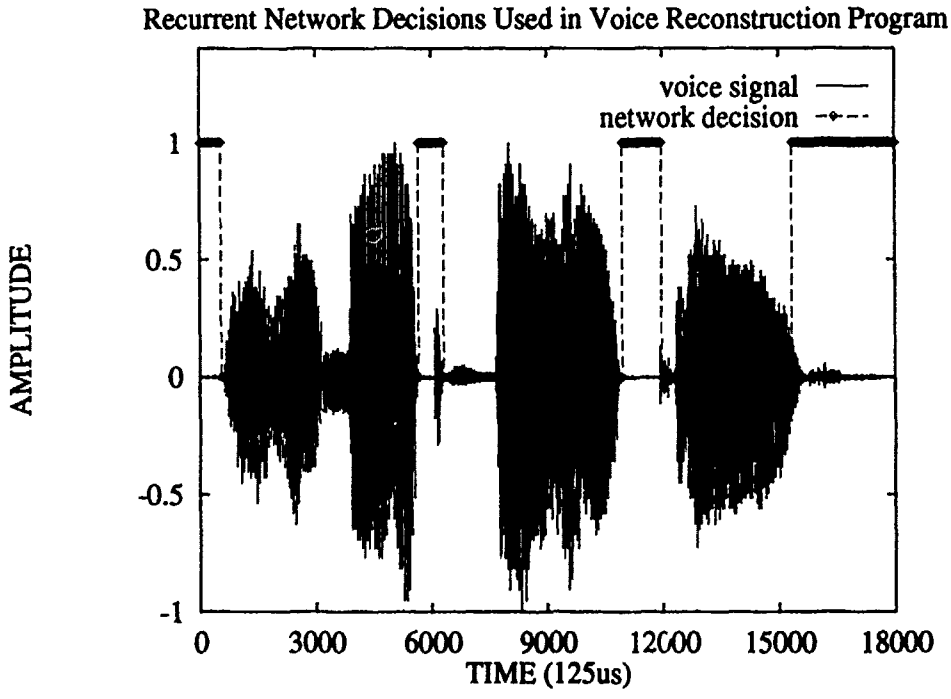


Figure 26. Recurrent network decisions made within the reconstruction program. Areas classified as "1" are considered fricatives (noisy regions), and noise was added to that portion of speech during reconstruction. No noise was added to the voiced regions classified a "0".

4.7 Summary

The recurrent neural network was tested using several temporally encoded data sets. From these test results, the network demonstrated the ability to learn the internal state problem and the second order IIR lowpass Butterworth filter problem. Specifically noted was the network's ability to learn both the temporal response and frequency domain response of the Butterworth filter by training only on the filter's impulse response.

The recurrent network was also tested on the classic XOR problem. However, it was discovered that the recurrent network did not learn this problem in the classic spatial sense. Rather it learns the problem in a spatio-temporal decision space. There was found no clear decision region which could be used to delineate the correct decisions from the incorrect decisions, as shown by Figure 9.

Following testing, the recurrent network was applied to two problems: head position tracking, and voice data reconstruction. The accuracy at which the network predicted the pilot's head position showed the recurrent network's ability to predict trajectories and motion as well as, or slightly better than, the best linear predictor. The application of the network to the reconstruction of voice data showed the network's ability to make accurate decisions based upon the learning of temporally encoded sequences. Thus, through both of these applications, the recurrent network displayed a high degree of generalization. Therefore, the extension of the recurrent neural network's application to a wide range of differing problems would be a straight-forward process.

V. Conclusions and Recommendations

This thesis effort has sought to encode the RTRL algorithm, test it, and use it to predict the future value of a function based upon the function's history. This process, called function prediction, is extremely important to many Air Force applications.

5.1 Conclusions

The RTRL algorithm has demonstrated the ability to learn several time dependent functions. From the test results outlined in Chapter IV, the network demonstrated the ability to learn the internal state problem, and the second order IIR lowpass Butterworth filter problem. Also the recurrent network demonstrated the ability to temporally learn the classic XOR problem.

However, in an exact sense, it could not learn the true spatial mapping of the XOR problem because of the te:nporal information contained in the sequential data. This was evidenced by the fact that no clear spatial decision region could be used to delineate the correct decisions from the incorrect decisions, as Figure 9 illustrates.

The recurrent network was also applied to two problems: head position tracking, and voice data reconstruction. The accuracy at which the network predicted the pilot's head positions showed the recurrent network's ability to predict trajectories and motion. The application of the network to the reconstruction of voice data showed the network's ability to learn temporally encoded sequences.

The recurrent network has demonstrated a high degree of accuracy as a function prediction tool. In the Butterworth filter application, the network not only simulated the response of the filter to various input signals, it did so predictively. In other words, the output of the network was a prediction of the output of the filter for the next time step. This prediction was based upon the current signal activation and the previous network response. For the head position tracking problem, the recurrent network demonstrated a

high degree of accuracy in predicting spatial head position at either 2 or 5 time steps in the future. The time step was arbitrary and was based on the sampling rate of the data being analyzed.

5.2 Recommendations

A recommendation that may improve the rate of convergence of the network entails the use of a network reduction scheme. One such method makes use of the Ruck saliency metric (16, 17). This method examines the responsiveness of the network's output to its input in order to rank the network's nodal usefulness. This way, the network can be pruned down, reducing the number of interconnection weights and processing nodes. A direct result of this reduction would be a great increase in computational speed and network convergence.

Another recommendation would be to investigate a way to determine what the recurrent neural network learned during the training process. It is known how the network learns and how it performs when trained, but it is still not exactly known what the network learns in order to make accurate decisions.

5.3 Future Research

Much more research is needed in the area of recurrent networks. Two specific topics are brought to light by this thesis. One is the concept of the recurrent network's capacity to remember. Does the network really remember the temporal nature of the task it is presented? Does it forget at some future time? How does the logistic squashing function affect the network's capacity to remember? The second topic of future research is the spatio-temporal mapping of the recurrent network's decision region. What kind of decision region does the recurrent network create in the process of making a decision? Do the internal state variables play an important role in the decision process? Is this decision region purely a temporal mapping, or does this mapping contain both spatial and temporal information?

Appendix A. *Software Development*

Appendix B contains the source listing for the modified RTRL algorithm developed at AFIT called "RECNET" (short for recurrent neural network). RECNET was written in "ANSI C" and has been successfully compiled and run on all of the following computer systems: Silicon Graphics 4D/GTX, Silicon Graphics Personal IRIS 4D, NeXT NeXTstation, and IBM/compatable "AT class" personal computers using *Turbo C++*. The main program file is named "recnet.c".

A.1 File Parameters

RECNET requires two data files, called "parameters.dat" and "data.dat" (default). "parameters.dat" is a data file which contains the following three numbers:

```
num_epochs learning_rate random_number_seed
```

The "num_epochs" (integer) is the epoch length for a specific training run. The "learning_rate" (float) is the learning rate of the network. If the output nodes of the network are defined as sigmoidal, the learning rate can be set to any value that works. If the output nodes are defined as linear, the learning rate must be small (alpha < 0.5) for the network to remain stable. The "random_number_seed" (integer) is the seed used to randomly generate the weight matrix. The data file "data.dat" is the default name for the data file to be read during training or testing. If a data filename is passed to RECNET at the command line, RECNET will read that filename as the input data file. For example, the command

```
recnet mydatafile.dat
```

will execute RECNET using "mydatafile.dat" as the current input data file. To test a data file, the command

```
recnet mytestfile.dat test
```

will execute RECNET using "mytestfile.dat" as the input test data file. The format of the data file is as follows:

```
numinputs numoutputs numnodes numvectors
in1 in2...numinputs  des1 des2...numoutputs  (vector1)
.   .     .          .    .     .            (vector2)
.   .     .          .    .     .            (vector3)
.   .     .          .    .     .            (vector4)
.   .     .          .    .     .            (vector5)
.   .     .          .    .     .            (vector6)
numvectors
```

where "numinputs" (integer) is the number of external inputs, "numoutputs" (integer) is the number of external outputs, "numnodes" (integer) is total number of processing nodes (which includes the output nodes), "numvectors" (integer) is the total number of input/desired_output vectors, "in1 in2..." (float) are the actual values to be read as inputs (there should be 'numinputs' of these), and "des1 des2..." (float) are the actual desired output values (there should be 'numoutputs' of these). Each vector is considered a seperate timed event.

A.2 Environment

RECNET dynamically configures itself using the data contained in the data file header line. During initialization, memory space is allocated for all variables and all inputs and desired outputs are read in before any computation begins. After initialization, the network begins training on the data, dynamically adjusting the weights until either the total error drops below 0.0005, or the total number of epochs have been reached.

RECNET will display different information depending on which mode of operation is selected. During training, RECNET will output to the terminal screen various information. First, it will show how it is configured by displaying the data file header line. Following

this, the epochwise total error is printed to show how the network is learning, or whether or not it is learning. During testing, RECNET outputs the current configuration to the terminal screen. In addition, it displays the names of the three data files it creates during the test run. These files are described in the next section.

A.3 Output

After network training is complete, several output data files are created. First, the data file "weights.dat" is created. It contains a $z(t)$ vector listing for the very last timed input vector and a listing of the complete weight matrix, in row-column format, after training. In addition, the files "desired.dat" and "netout.dat" are created. The file "desired.dat" contains a listing of the desired output values, and the file "netout.dat" contains a listing of the actual network output values corresponding to the appropriate desired output value. These two files are separate to aid in plotting the data.

After testing, RECNET creates three data files. They are described as follows: "testcheck.dat" contains a comparative listing of the computed network output and the desired network output, "testdes.dat" contains a listing of the desired network output, and "testout.dat" contains a listing of the computed netork output. Again, these files are seperate to aid in plotting the data.

Appendix B. *Recurrent Neural Network Source Code*

This appendix contains a listing of the modified real-time recurrent learning algorithm source code and its supporting functions. The files "nrutil.c" and "ran1.c" were used from the *Numerical Recipies in C* book (12).

```
/* definitions.h ***************************************************

    File containing function declarations and variable
    declarations for the main program called recnet.c.

  date:  30 May 91
  written by:  Randall L. Lindsey

*****************************************************************/
float *vector();
float **matrix();
float ***matrix3d();
float ran1();

FILE *ifp, *ofp;
int run=1;
char str[80], *datafile;
int nrows, ncols, i, j, k, l, m, n;
int epochs, a, t;
int num_inputs, num_outputs, num_nodes, num_vectors;
int seed, idum=1;
float alpha, J[2], sum, kron;
float *y, *s, *e, *yprime;
float **z, **d, **w, **delw;
float ***p, ***p_old, ***p_temp;
float sigmoid(float x);
void init_net();
void train_net();
void test_net();
void read_data();
void propagate();
void compute_output();
void compute_error();
void update();
void reset_delw_s();
void reset_p();
void save_weights();
void read_weights();
```

```
void check_file();

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

/*** MACROS.H ****************************************************/

char junk_response[256];

#define fskip_line(A) fgets(junk_response, 256, A)
#define skip_line gets(junk_response)

#define rloopi(A) for(i=(A)-1;i≥0;i--)
#define rloopj(A) for(j=(A)-1;j≥0;j--)
#define rloopk(A) for(k=(A)-1;k≥0;k--)
#define rloopl(A) for(l=(A)-1;l≥0;l--)
#define rloopij(A,B) for(i=(A)-1;i≥0;i--) for(j=(B)-1;j≥0;j--)

#define loopi(A) for(i=0;i<A;i++)
#define loopj(A) for(j=0;j<A;j++)
#define loopk(A) for(k=0;k<A;k++)
#define loopl(A) for(l=0;l<A;l++)
#define loopij(A,B) for(i=0;i<A;i++) for(j=0;j<B;j++)

#define CREATE_FILE(A,B,C) if((A=fopen(B,"w")) == NULL) { \
     printf(strcat(C,": can't open for writing - %s.\n"),B); \
   exit (-1); }
#define OPEN_FILE(A,B,C) if((A=fopen(B,"r")) == NULL) { \
     printf(strcat(C,": can't open for reading - %s.\n"),B); \
   exit (-1); }

#define IABS(A) ((int)((-(A)<(A))?((A)):(-(A))))


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



/* RECNET.C ******************************************************

     A recurrent neural network which follows the algorithm
     proposed by Williams and Zipser in their paper "A Learning
     Algorithm for Continually Running Fully Recurrent
     Neural Networks", Neural Computation 1, 270-280 (1989).

   date:    30 May 91
   update:  15 Jul 91
```

```
   written by:  Randall L. Lindsey, GEO-91D
*************************************************************************/

#include <stdio.h>
#include "macros.h"
#include <math.h>
#include "definitions.h"
#include <string.h>

void main(int argc, char *argv[])
{
 switch (argc) {
 case 1:
  datafile="data.dat"; /* Default name of datafile.                */
  check_file();   /* Check to see if the datafile name exists.   */
  init_net();   /* Initialize and define all network variables.
          Allocate memory for all vectors and matricies
          and set initially to zero.  Randomly set the
          weight matrix using the pseudo-random number
          generator.                                               */
  read_data();   /* Read data vector array and desired output.   */
  train_net();   /* Propagate inputs and update weights based on
          gradient descent.                                        */
    break;

 case 2:
  datafile=argv[1];  /* User specified name of datafile.          */
  check_file();   /* Check to see if the datafile name exists.   */
  init_net();   /* Initialize and define all network variables.
          Allocate memory for all vectors and matricies
          and set initially to zero.  Randomly set the
          weight matrix using the pseudo-random number
          generator.                                               */
  read_data();   /* Read data vector array and desired output.   */
  train_net();   /* Propagate inputs, compute outputs, and
          update weights based on gradient descent.   */
  break;

 case 3:
  datafile=argv[1];  /* User specified name of datafile.          */
  check_file();   /* Check to see if the datafile name exists.   */
  init_net();   /* Initialize and define all network variables.
          Allocate memory for all vectors and matricies
          and set initially to zero.  Randomly set the
          weight matrix using the pseudo-random number
          generator.                                               */
  test_net();   /* Propagate inputs and compute outputs.         */
```

```
  break;

 default:
  printf("\nUsage:  net [datafilename.dat] [testflag]\n\n");
  break;
 }

} /* End MAIN() of RECNET.C */


void train_net()  /* Written 10 Jun 91, RLL. */
{
 /* Begin main loop portion */

  ofp=fopen("error.trn", "w");
  for(a=0;a<epochs;a++) {
   J[0] = J[1];
   J[1] = 0.;
   for(t=0;t<num_vectors;t++) {


  propagate();    /* Computes the state of the net at time t.
          Store previous outputs y[t-1] as part of
          the new input vector z[t][i].  Sum all
          z[][]*w[][] inputs into the activation
          vector s[t] for input into y[t].        */

  compute_error(); /* Computes the error at time t.
          How far off are the outputs from the
                desired values?  Compute total error.    */

  compute_output(); /* Compute the output y(t+1)=f[s(t)].       */

  update();    /* Computes del_w(t), and p(t+1).  Backprop
          error through net and perform gradient
          descent to calculate the delta weights.  */

  reset_delw_s(); /* Reset delta weights and s[t] vectors
                to zero for the next iteration.         */
   }

   printf("%s % f\n","total error =",J[1]); /* Print total error.*/
   if ((a > 5) && (J[0]/J[1] < 0.95)) {
   alpha = alpha/2.0;
   printf("% f % f alpha = % f\n",J[0],J[1],alpha);
   }
```

```
    if (J[1] < 0.0005) {   /* If total error is less than a specific */
    save_weights();  /* fractional value (arbitrary), then exit.*/
    printf("%d\n",a);
    exit(0);
     }

     fprintf(ofp,"% f\n",J[1]);

     reset_p();       /* Zero p_old[][][] matrix for next epoch.   */

    } /* End main loop portion */

    fclose(ofp);
    save_weights();      /* Save weights, input vector z, and desired
              output to a data file for future use.      */

    return;

} /* end function train_net() */


void test_net()      /* Written 10 Jun 91, RLL. */
{
 /* Begin main loop portion */

    read_weights(); /* Read weight matrix and saved p states.        */
    read_data(); /* Read data vector array and desired output.    */

    ofp=fopen("error.tst", "w");
    J[1] = 0.;
    for(t=0;t<num_vectors;t++) {

    propagate();     /* Store previous outputs y[t-1] as part of
              the new input vector z[t][i].  Sum all
              z[][]*w[][] inputs into the activation
              vector s[t] for input into y[t].          */

    compute_output(); /* Compute the output y(t+1)=f[s(t)].          */

    compute_error(); /* Computes the error at time t.
              How far off are the outputs from the
                   desired values?  Compute total error.      */

    reset_delw_s(); /* Reset delta weights,and s[] vectors to
              zero for the next iteration.              */

      fprintf(ofp,"% f\n",J[1]);
```

```
} /* End main loop portion */

fclose(ofp);

ofp=fopen("testcheck.dat", "w");
loopi(num_vectors) {
loopj(num_outputs)
  fprintf(ofp,"% f ",z[i][j+m]);
loopj(num_outputs)
  fprintf(ofp,"% f ",d[i][j]);
fprintf(ofp,"\n");
}
fclose(ofp);

ofp=fopen("testdes.dat", "w");
loopi(num_vectors)
loopj(num_outputs)
  fprintf(ofp,"% f\n",d[i][j]);
fclose(ofp);

ofp=fopen("testout.dat", "w");
loopi(num_vectors)
loopj(num_outputs)
  fprintf(ofp,"% f\n",z[i][j+m]);
fclose(ofp);


printf("'testcheck.dat' contains test data.\n");
printf("'testout.dat' contains net output test data.\n");
printf("'testdes.dat' contains desired output test data.\n");
return;

} /* end function test_net() */


float sigmoid(float x)      /* Written 30 May 91, RLL. */
{
  static float max_val=50.;

  if (x > max_val)
    return 1.0;
  if (x < -max_val)
    return 0.0;
  return 1/(1 + exp(-x));

} /* end sigmoid */
```

```
void init_net()      /* Written 10 Jun 91, RLL. */
{

 /* Read data from the input file "parameters.dat" */

  ifp=fopen("parameters.dat","r");
  fscanf(ifp,"%d %f %d",&epochs,&alpha,&seed);
  fclose(ifp);

 /* Read data from the input file datafile (user specified) */

  ifp=fopen(datafile,"r");
  fscanf(ifp,"%d %d %d",&num_inputs,&num_outputs,&num_nodes);
  fscanf(ifp,"%d",&num_vectors);
  printf("%d %d %d\n",num_inputs,num_outputs,num_nodes);
  fclose(ifp);

  m = num_inputs + 1;  /* # of external inputs */
  nrows = n = num_nodes;  /* # of rows for weight matrix */
  ncols = m + num_nodes;  /* # of cols for weight matrix */

 /* Allocate memory for vectors and matrices */

  e=vector(0,nrows-1);  /* error vector */
  y=vector(0,nrows-1);  /* output vector */
  s=vector(0,nrows-1);  /* sum of weighted inputs */
  yprime=vector(0,num_nodes-1);  /* dy/dw */
  w=matrix(0,nrows-1,0,ncols-1);      /* weight matrix */
  delw=matrix(0,nrows-1,0,ncols-1);   /* delta weights */
  z=matrix(0,num_vectors,0,ncols-1);  /* input vector array */
  d=matrix(0,num_vectors,0,ncols-1);  /* desired output array */
  p=matrix3d(0,nrows-1,0,ncols-1,0,nrows-1);  /* dy/dw */
  p_old=matrix3d(0,nrows-1,0,ncols-1,0,nrows-1);  /* dy/dw */

 /* Initialize variables to zero */

  J[0]=J[1]=0.0;
  loopij(num_vectors,ncols)
    z[i][j] = 0.;
  loopij(num_vectors,num_outputs)
    d[i][j] = 0.;
  loopi(nrows) {
    y[i] = s[i] = e[i] = 0.;
    loopj(ncols) {
      w[i][j] = delw[i][j] = 0.;
```

```
      loopk(nrows)
        p[i][j][k] = p_old[i][j][k] = 0.;
    }
  }

 /* Initialize weight matrix using psuedo-random numbers */

  idum = -IABS(seed);
  ran1(&idum);
  loopi(nrows) {
    loopj(ncols) {
      w[i][j] = 2*ran1(&idum)-1.0;
      printf("% f    ",w[i][j]);
    }
    printf("\n");
  }

 /* Initialize first input to 1 (non-external) */

  loopi(num_vectors)
    z[i][0] = 1.;

  return;
}


void read_data()   /* Written 10 Jun 91, RLL. */
{
 /* Read data file external inputs */

  ifp=fopen(datafile, "r");
  fskip_line(ifp);
  loopi(num_vectors) {
    loopj(num_inputs)
      fscanf(ifp,"%f",&z[i][j+1]);
    loopj(num_outputs)
      fscanf(ifp,"%f",&d[i][j]);
  }
  fclose(ifp);
  return;
}


void propagate()   /* Written 10 Jun 91, RLL. */
        /* Computes the state of the net at time t, and
           initializes the z vector for time t.          */
{
```

```
/* Set previous outputs y[k]=y(t) as part of the next input z[t][k+m]. */

  loopk(nrows)
    z[t][k+m] = y[k];

/* Sum all inputs into each of the k nodes. */

  loopk(nrows)
    loopi(ncols)
      s[k] += w[k][i] * z[t][i];

  return;
}


void compute_output()   /* Written 16 Jul 91, RLL. */
        /* Computes the output at time (t+1), ie y(t+1). */
{
 /* Process each of the k nodes as Sigmoidal functions with input s[t]
    unless LINEAR is defined, in which only output nodes are linear
    functions of s[t] and the remaining hidden nodes remain Sigmoidal.
    The output computed is y[k] = y(t+1) = f(s[t]).
 */

#ifdef LINEAR
   loopk(num_outputs)
     y[k] = s[k];
   loopk(nrows-num_outputs)
     y[k+num_outputs] = sigmoid(s[k+num_outputs]);
#else
   loopk(nrows)
     y[k] = sigmoid(s[k]);   /* Here, y[k]=y(t+1). */
#endif

   return ;
}


void compute_error()   /* Written 10 Jun 91, RLL. */
{

 /* Compute error at time t based on desired output values.  Returns a
    zero error for t=0 on first epoch. */

   if ((t == 0) && (a == 0)) return;
   else
```

```
    loopk(num_outputs)
      e[k] = d[t][k] − y[k];

  /* Total error cumulated over each epoch.  After each epoch, J = 0.  */

    loopk(num_outputs)
      J[1] += 0.5 * e[k] * e[k];

    return ;
}


void update()      /*  Written 10 Jun 91, RLL.
          Modified 28 Jun 91, RLL.  */
{
  /* Compute change of weights at time t. delw is reset to zero at each
     iteration (time step), and p_old is p(t).  */

    loopij(nrows,ncols)
      loopk(num_outputs)
        delw[i][j] += alpha * e[k] * p_old[i][j][k];

  /* Update rules.  Computes p(t+1).  */

#ifdef LINEAR
    loopk(num_outputs)
      yprime[k] = z[t][k];
    loopk(nrows−num_outputs)
      yprime[k+num_outputs] = y[k]*(1.0−y[k]);
#else
    loopk(nrows)
      yprime[k] = y[k]*(1.0−y[k]);      /* Uses y[k] = y(t+1).      */
#endif
                  /* m = num_inputs + 1      */
    loopi(nrows)              /* nrows = num_nodes.      */
      loopj(ncols)            /* ncols = num_nodes +     */
        loopk(nrows) {          /*       num_inputs + 1  */
          kron = 0.0;         /* Kronecker delta function.*/
          if (i==k) kron = 1.0;

          sum = 0.;
          loopl(num_nodes)
        sum += w[k][l+m]*p_old[i][j][l];      /* p_old = p(t).  */

          p[i][j][k] = yprime[k]*(sum+kron*z[t][j]);  /* Uses z(t).    */

        } /* p[][][] is now for time p(t+1).  */
```

```
 /* Update weights.  Computes weights for time w(t+1). */

  loopij(nrows,ncols)
      w[i][j] += delw[i][j];

 /* Save partial derivitives for next iteration (time t+1) and reset
    p matrix by swapping the pointers of the old p matrix with the new
    p matrix.  */

  p_temp = p_old;
  p_old = p;       /* p_old is now p(t+1). */
  p = p_temp;

  return ;
}


void reset_delw_s()   /*  Written 30 May 91, RLL.  */
{

 /* Reset delta weights and input sum to zero for next calculation. */

  loopij(nrows,ncols)
    delw[i][j] = 0.;
  loopi(nrows)
    s[i] = 0.;
      return;
}

void reset_p()   /*  Written 15 Jul 91, RLL.  */
{

 /* Zero p_old[][][] for next calculation. */

  loopij(nrows,ncols)
    loopk(nrows)
     p_old[i][j][k] = 0.;
  loopi(nrows)
    y[i] = 0.;
  return;
}


void save_weights()   /*  Written 28 Jun 91, RLL.  */
{
  FILE *afp;
```

```
  ofp=fopen("weights.dat", "w");
  i = num_vectors - 1;
    loopj(ncols)
      fprintf(ofp,"% f   ",z[i][j]);
  fprintf(ofp,"\n");
  loopi(nrows) {
    loopj(ncols)
      fprintf(ofp,"% f   ",w[i][j]);
    fprintf(ofp,"\n");
  }
  fclose(ofp);
  afp=fopen("netout.dat", "w");
  loopi(num_vectors) {
    loopj(num_outputs)
      fprintf(ofp,"% f  ",z[i][j+m]);
    fprintf(ofp,"\n");
  }
  fclose(afp);
  afp=fopen("desired.dat", "w");
  loopi(num_vectors) {
    loopj(num_outputs)
      fprintf(ofp,"% f  ",d[i][j]);
    fprintf(ofp,"\n");
  }
  fclose(afp);
  return;
}

void read_weights()  /* Written 28 Jun 91, RLL. */
{
  ifp=fopen("weights.dat", "r");
  i = 0;
    loopj(ncols)
      fscanf(ifp,"%f",&z[i][j]);
  loopi(nrows)
    loopj(ncols)
      fscanf(ifp,"%f",&w[i][j]);
  fclose(ifp);
  return;
}

void check_file()  /* Written 10 Jul 91, RLL. */
{
  FILE *afp;

  afp = fopen(datafile,"r");
```

```
  if(afp == NULL) {
    /*strcpy(afile,"File not found");*/
    printf("\n%s %s\n",datafile,": File not found.");
    exit(0);
  }
  else fclose(afp);
  return;
}

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

/* NRUTIL.C  ****************************************

   Utilities which create vectors, matricies, and
   3-D matrices.

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

#include "malloc.h"
#include <stdio.h>

void nrerror(error_text)
char error_text[];
{
  void exit();

  fprintf(stderr,"Numerical Recipes run-time error...\n");
  fprintf(stderr,"%s\n",error_text);
  fprintf(stderr,"...now exiting to system...\n");
  exit(1);
}

float *vector(nl,nh)
int nl,nh;

{
  float *v;

  v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
  if (!v) nrerror("allocation failure in vector()");
  return v-nl;
}

int *ivector(nl,nh)
int nl,nh;
{
  int *v;
```

```
  v=(int *)malloc((unsigned) (nh-nl+1)*sizeof(int));
  if (!v) nrerror("allocation failure in ivector()");
  return v-nl;
}

double *dvector(nl,nh)
int nl,nh;
{
  double *v;

  v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
  if (!v) nrerror("allocation failure in dvector()");
  return v-nl;
}


float **matrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
  int i;
  float **m;

  m=(float **) malloc((unsigned) (nrh-nrl+1)*sizeof(float*));
  if (!m) nrerror("allocation failure 1 in matrix()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
    m[i]=(float *) malloc((unsigned) (nch-ncl+1)*sizeof(float));
    if (!m[i]) nrerror("allocation failure 2 in matrix()");
    m[i] -= ncl;
  }
  return m;
}

/*************************************************************
  matrix3d() created by Randall Lindsey on 15 May 91 for
  use in recnet.c
*************************************************************/

float ***matrix3d(nrl,nrh,ncl,nch,ndl,ndh)
int nrl,nrh,ncl,nch,ndl,ndh;
{
  int i,j;
  float ***m;

  m=(float ***) malloc((unsigned) (nrh-nrl+1)*sizeof(float**));
```

```
  if (!m) nrerror("allocation failure 1 in matrix3d()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
    m[i]=(float **) malloc((unsigned) (nch-ncl+1)*sizeof(float*));
    if (!m[i]) nrerror("allocation failure 2 in matrix3d()");
    m[i] -= ncl;
    for(j=ncl;j<=nch;j++) {
      m[i][j]=(float *) malloc((unsigned) (ndh-ndl+1)*sizeof(float));
      if (!m[i][j]) nrerror("allocation failure 3 in matrix3d()");
      m[i][j] -= ndl;
    }
  }
  return m;
}

double **dmatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
  int i;
  double **m;

  m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
  if (!m) nrerror("allocation failure 1 in dmatrix()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
    m[i]=(double *) malloc((unsigned) (nch-ncl+1)*sizeof(double));
    if (!m[i]) nrerror("allocation failure 2 in dmatrix()");
    m[i] -= ncl;
  }
  return m;
}

int **imatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
  int i,**m;

  m=(int **)malloc((unsigned) (nrh-nrl+1)*sizeof(int*));
  if (!m) nrerror("allocation failure 1 in imatrix()");
  m -= nrl;

  for(i=nrl;i<=nrh;i++) {
    m[i]=(int *)malloc((unsigned) (nch-ncl+1)*sizeof(int));
    if (!m[i]) nrerror("allocation failure 2 in imatrix()");
    m[i] -= ncl;
```

```
    }
    return m;
}

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

/* RAN1.C ***********************************************

   Numerical Recipies pseudo-random number generator.

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

#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

extern float ran1(idum)
int *idum;
{
    static long ix1,ix2,ix3;
    static float r[98];
    float temp;
    static int iff=0;
    int j;
    void nrerror();

    if (*idum < 0 || iff == 0) {
        iff=1;
        ix1=(IC1-(*idum)) % M1;
        ix1=(IA1*ix1+IC1) % M1;
        ix2=ix1 % M2;
        ix1=(IA1*ix1+IC1) % M1;
        ix3=ix1 % M3;
        for (j=1;j≤97;j++) {
            ix1=(IA1*ix1+IC1) % M1;
            ix2=(IA2*ix2+IC2) % M2;
            r[j]=(ix1+ix2*RM2)*RM1;
        }
        *idum=1;
```

```
    }
    ix1=(IA1*ix1+IC1) % M1;
    ix2=(IA2*ix2+IC2) % M2;
    ix3=(IA3*ix3+IC3) % M3;
    j=1 + ((97*ix3)/M3);
    if (j > 97 || j < 1) nrerror("RAN1: This cannot happen.");
    temp=r[j];
    r[j]=(ix1+ix2*RM2)*RM1;
    return temp;
}

#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
#undef M3
#undef IA3
#undef IC3

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

/*************************************************************/
```

The following additional listing is a supporting data file required for the recurrent network program to work properly.

```
/* PARAMETERS.DAT ***********************************************/

300 4.0 987654321

/*************************************************************/
```

Appendix C. *Source Code for Creation of Data*

This appendix contains a listing of the source code which generated the data for testing the modified recurrent neural network.

```
/* MAKE_DATA.C ***********************************************/


#include <stdio.h>
#include "macros.h"
#include "ran1.c"

FILE *ofp, *ifp;

void main(int argc, char *argv[])
{
  float class,junk[2][2048],x[2][2048],a0,a1,a2,b0,b1;
  int idum=1,i,j,bubba;

 switch (argc) {
 case 1:
 case 2:
 printf("\n%s\n\n","Usage: make_data filename.dat num_nodes [bin]");
  exit(0);
  break;

 case 3:
  a0=0.0676; a1=0.1352; a2=0.0676;
  b0=1.1422; b1=-0.4124;
  bubba=atoi(argv[2]);
  ofp=fopen(argv[1], "w");
  idum = -IABS(737496732);
  ran1(&idum);
  fprintf(ofp,"%d %d %d %d\n",1,1,bubba,100);
  x[1][0]=x[1][1]=0.0;
  loopi(100)
    x[0][i] = 2.0*r_.1(&idum)-1.0;
  loopi(98)
     x[1][i+2]=a0*x[0][i+2]+a1*x[0][i+1]+a2*x[0][i]+b0*x[1][i+1]+b1*x[1][i];
  loopi(100)
    fprintf(ofp,"% f % f \n",x[0][i],x[1][i]);
  fclose(ofp);
```

```
  break;

 case 4:
  bubba=atoi(argv[2]);
  ofp=fopen(argv[1], "w");
  idum = -IABS(97475298);
  ran1(&idum);
  fprintf(ofp,"%d %d %d %d",1,1,bubba,2048);
  junk[1][0]=0.;
  loopi(2048)
    junk[0][i]=ran1(&idum);
  loopi(2048)
    junk[1][i]=junk[0][i];
  loopi(2048)
    fprintf(ofp,"\n% f % f ",junk[0][i],junk[1][i]);
  fclose(ofp);
  break;
 }
}


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

/* XOR_DATA.C ***************************************/


#include <stdio.h>
#include "macros.h"
#include "ran1.c"

FILE *ofp, *ifp;

void main(int argc, char *argv[])
{
   float class,junk[2][1024],seed;
   int idum=1,i,j,bubba;

 switch (argc) {
 case 1:
 case 2:
 case 3:
 printf("\n%s\n\n","Usage: make_data filename.dat num_nodes seed
[bin]");
   exit(0);
   break;

 case 4:
```

```
    bubba=atoi(argv[2]);
    ofp=fopen(argv[1], "w");
    idum = -IABS(seed);
    ran1(&idum);
    fprintf(ofp,"%d %d %d %d",2,1,bubba,1024);
    loopi(1024) {
      loopj(2) {
        junk[j][i]=ran1(&idum);
        if (junk[j][i]>0.5) junk[j][i]=1.0;
        else junk[j][i]=0.0;
      }
      if (i < 2) class=1.0;
      else {
        if ((junk[0][i-2]>0.5) && (junk[1][i-2]>0.5)) class=0.0;
        if ((junk[0][i-2]≤0.5) && (junk[1][i-2]>0.5)) class=1.0;
        if ((junk[0][i-2]>0.5) && (junk[1][i-2]≤0.5)) class=1.0;
        if ((junk[0][i-2]≤0.5) && (junk[1][i-2]≤0.5)) class=0.0;
      }
      fprintf(ofp,"\n%f %f %f ",junk[0][i],junk[1][i],class);
    }
    fclose(ofp);
    break;

  case 5:
    bubba=atoi(argv[2]);
    ofp=fopen(argv[1], "w");
    idum = -IABS(seed);
    ran1(&idum);
    fprintf(ofp,"%d %d %d %d",2,1,bubba,1024);
    loopi(1024) {
      loopj(2)
        junk[j][i]=ran1(&idum);
      if (i < 2) class=1.0;
      else {
        if ((junk[0][i-2]>0.5) && (junk[1][i-2]>0.5)) class=0.0;
        if ((junk[0][i-2]≤0.5) && (junk[1][i-2]>0.5)) class=1.0;
        if ((junk[0][i-2]>0.5) && (junk[1][i-2]≤0.5)) class=1.0;
        if ((junk[0][i-2]≤0.5) && (junk[1][i-2]≤0.5)) class=0.0;
      }
      fprintf(ofp,"\n%f %f %f ",junk[0][i],junk[1][i],class);
    }
    fclose(ofp);
    break;
  }
}

/*****************************************************************/
```

Appendix D. *Utility Source Code*

This appendix contains a listing of the utilities source code. These programs were used to make the data better suited to the neural network environment.

```
/** STAT-NORM.C ***********************************************

   Performs statistical normalization on filename.dat and
   creates filename.dat.sn as its output.

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


#include <stdio.h>
#include <math.h>

#define INPUTS   75   /* max number of features */

/** begin Main Program **/

void main (argc, argv)
int argc;
char *argv[];
{
   /*=======================================================*/
   /*=== local variables                          ===*/
   /*=======================================================*/

   FILE *fopen();
   FILE *input, *fopen();
   char infile[50];
   FILE *output;
   char outfile[50];
   float value, trash;
   float deviation[INPUTS], average[INPUTS];
   int i, j, inputs, outputs, ivalue;
   int count1, count2, count, waste, temp;

   /*=======================================================*/
   /*=== did user specify an input file          ====*/
   /*=======================================================*/

   if (argc != 2)  {
     printf("\n\nUsage -> stat-norm <filename>\n\n");
```

```
  /*=== exit after pointing out the error ===*/
  exit (1000);
}

/*=========================================================*/
/*=== user did specify an input file              ===*/
/*=========================================================*/

strcpy (infile, argv[1]);   /* use inputted name as base */

/*=========================================================*/
/*=== Open Input File                            ===*/
/*=========================================================*/

printf ("\nOpening Input File:  %s\n\n", infile);

if (!(input = fopen(infile, "rb")))
{
  printf ("\nCan't open input file: %s\n\n", infile);
  exit (2000);
}

/*=========================================================*/
/*=== read the header information                ===*/
/*=========================================================*/

fscanf (input, "%d %d %d %d\n", &count1, &outputs, &inputs, &count);

if (count1 < 0 || count2 < 0 || inputs < 0 || outputs < 0)
{
  printf ("One of the header inputs is negative\n\n");
  exit (3000);
}

printf ("There are %d training vectors\n", count);
printf ("There are %d test vectors\n", count);
printf ("There are %d inputs\n", inputs);
printf ("There are %d outputs\n\n", outputs);

/*count = count1 + count2;*/

/*=========================================================*/
/*=== initialize things                          ===*/
/*=========================================================*/

for (i = 0; i < inputs; i++)
{
```

```
  average[i] = deviation[i] = 0.0;
}

/*=============================================================*/
/*=== loop until all data has been read in          ===*/
/*=============================================================*/

printf ("Reading the Data\n\n");

for (i = 0; i < count; i++)
{
  /*fscanf (input, "%d ", &trash);*/   /* read line counter */

  for (j = 0; j < inputs; j++)
  {
    fscanf (input, "%f  ", &value);   /* read float values */

    average[j] += value;
  }

/*   for (j = 0; j < outputs-1; j++)
  {
    fscanf (input, "%f ", &value);
  } */

  /*fscanf (input, "%f\n\n", &value); */
}

fclose (input);

/*=============================================================*/
/*=== calculate the averages                        ===*/
/*=============================================================*/

printf ("Calculating Averages\n\n");

for (i = 0; i < inputs; i++)
{
  average[i] /= (float)count;
}

/*=============================================================*/
/*=== Re-open the input file                        ===*/
/*=============================================================*/

printf ("Re-Opening Input File:  %s\n\n", infile);
```

```
if (!(input = fopen(infile, "rb")))
{
  printf ("\nCan't re-open input file: %s\n\n", infile);
  exit (2000);
}

/*=============================================================*/
/*=== throw away the header information this time ===*/
/*=============================================================*/

fscanf (input, "%d %d %d %d\n", &waste, &waste, &waste, &waste);

/*=============================================================*/
/*=== loop until all data has been read in       ===*/
/*=============================================================*/

printf ("Reading the Data\n\n");

for (i = 0; i < count; i++)
{
  /*fscanf (input, "%d ", &trash);*/   /* read line counter */

  for (j = 0; j < inputs; j++)
  {
    fscanf (input, "%f  ", &value);   /* read float values */

    value -= average[j];   /* subtract off the average */
    value *= value;        /* square the result */

    deviation[j] += value; /* hang onto it until all done */
  }

/*  for (j = 0; j < outputs-1; j++)
  {
    fscanf (input, "%f ", &value);
  } */

  /*fscanf (input, "%f\n\n", &value);*/
}

fclose (input);

/*=============================================================*/
/*=== calculate the standard deviation           ===*/
/*=============================================================*/

printf ("Calculating Standard Deviations\n\n");
```

```
for (i = 0; i < inputs; i++)
{
   deviation[i] /= count - 1;
   deviation[i]  = (float)sqrt((double)deviation[i]);
}

/*=============================================================*/
/*=== make output-file name                              ===*/
/*=============================================================*/

sprintf (outfile, "%s.sn", argv[1]);

/*=============================================================*/
/*=== Open Output File                                   ===*/
/*=============================================================*/

printf ("Opening Output File:  %s\n\n", outfile);

if (!(output = fopen(outfile, "wb")))
{
   printf ("\nCan't open output file: %s\n\n", outfile);
   exit (2000);
}

/*=============================================================*/
/*=== Re-open the input file                             ===*/
/*=============================================================*/

printf ("Re-Opening Input File (last time):  %s\n\n", infile);

if (!(input = fopen(infile, "rb")))
{
   printf ("\nCan't re-open input file: %s\n\n", infile);
   exit (2000);
}

/*=============================================================*/
/*=== read and save header                               ===*/
/*=============================================================*/

fscanf (input, "%d %d %d %d\n", &count1, &outputs, &inputs, &count);
fprintf (output, "%d %d %d %d\n", count1, outputs, inputs, count);

/*=============================================================*/
/*=== read data in, modify it, save it back out          ===*/
/*=============================================================*/
```

```
  printf ("Reading, Modifying and Re-Saving the Data\n\n");

  for (i = 0; i < count; i++)
  {
    /*fscanf (input, "%d ", &ivalue);*/   /* read line counter */
    /*fprintf (output, "%d ", ivalue);*/   /* save line counter */

    for (j = 0; j < inputs; j++)
    {
      fscanf (input, "%f ", &value); /* read float value */

      value -= average[j];        /* modify the value */
      value /= deviation[j];

      fprintf (output,"%f ",value); /* save modified value */
    }
    fprintf (output,"\n");
  /*  for (j = 0; j < outputs-1; j++)
    {
      fscanf (input, "%f ", &value);
      fprintf (output, "%f ", value);
    } */

    /*fscanf (input, "%d\n\n", &ivalue);
    fprintf (output, "%d\n\n", ivalue);*/
  }

  fclose (input);
  fclose (output);

  /*=========================================================*/
  /*=== we're done                                      ===*/
  /*=========================================================*/

  printf ("Finished.\n\n");
}

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

/** FFT.C ***************************************************

   Fast Fourier Transform Program:

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

#include <stdio.h>
```

```
#include <math.h>

#define loopi(A) for(i=0;i<(A);i++)
#define loopj(A) for(j=0;j<(A);j++)
#define loopij(A,B) for (i=0; i<(A); i++)\
for (j=0; j<(B); j++);

#define SQ(A)  (A*A)
#define PI   3.1415926

main(int argc,char *argv[])
{
  FILE *fin, *fout;
  float *output,*input,*trunc_out;
  float norm;
  float  *vector();
  /*void doflip();*/
  void fourn();
  /*void truncate();*/
  /*void *free_vector();*/
  char name[30];
  int i,j, nn[1], ndim, isign, new_order, order, image_size;
  if(argc ≠ 3) {
     printf("!!! The command line should be !!!:\n\n  fft_trunc
infile outfile \n\n");
    exit(0);
  }

  printf("!!!  Input the input images SIZE and ORDER: ");
  scanf("%d%d",&image_size,&order);

/*******************set up dynamic allocation*****************/

  input = vector(0,2*image_size*image_size-1);
  output = vector(0,image_size*image_size-1);



/******************* Set Up Files ************************/

  if ((fin=fopen(argv[1],"r")) == NULL) {
   printf("I can't open the input file");
   exit(-1);
  }

  if ((fout=fopen(argv[2],"w")) == NULL){
```

```
    printf("I can't open the output file");
    exit(-1);
    }

/**************Read File ********************/

   loopi(2*image_size*image_size-1)   /* initialize array to zero */
    input[i] = 0.0;

   loopi(image_size*image_size-1)    /*read data in the fourn format */
    fscanf(fin, "%f\n", &input[i*2]); /* see numerical recipes in c */

   fclose(fin); /*close input file */

/***** Initialization parameters for FFT *******/

   nn[0]=image_size; /* size of mput IAW fourn()  */
   nn[1]=image_size;

   ndim=1;          /* one dim FFT  */
   /*ndim=2; */       /* two dim FFT  */
   isign=1;         /* FFT */

   fourn(input-1,nn-1,ndim,isign);



/********** Find Fourier Magnitude **********/

   j=0;
   for(i=0;i<(2*image_size*image_size-1); i+=2) {
     output[j]=sqrt((double)SQ(input[i])+(double)SQ(input[i+1]));
     j++;
   }

   norm=output[0]; /* d.c component used for normalization **/

   printf("%4.0f\n",norm);

/*****  normalize and write output of FFT in argv[2] file  ***/

   loopi(image_size*image_size) {
     output[i]=output[i]/norm;
     fprintf(fout, "%1.4f\n", output[i]);
   }

   fclose(fout);
```

```
/***** doflip*******************************/

  /*doflip(output,image_size); */ /* converts fourn format to human format */
  /*printf("%4.4f\n",output[8128]);*/

/******* truncate  ********************************************
 truncate takes fft(output) of size(image_size) and truncates the
 FFT to order specified plus d.c.  the array is returned in
 trunc_out, the argv[2] is used as a header when truncate writes
 the output in netfft.dat
*************************************************************/

  if(order != 0){
  new_order = 2*order+1;
  trunc_out = vector(0,image_size*image_size-1);
  truncate(output,image_size,order,trunc_out, argv[2]);
  free_vector(trunc_out,0,image_size*image_size-1);
  }

  free_vector(input,0,2*image_size*image_size-1);
  free_vector(output,0,image_size*image_size-1);


}

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

/*************************************************************
NAME: fourn.c
DESCRIPTION:  Numerical Recipies multi dimensional FFT routine.
Requires a complex column vector  as follows:
/ real a(1)/
/ complex a(1)/
/ real a(2)/
/ complex a(2)/
/ etc/
SUBROUTINES CALLED:
WRITTEN BY: Numerical Recipies in C

*************************************************************/
#include <math.h>

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

void fourn(data,nn,ndim,isign)
float data[];
```

```
int nn[],ndim,isign;
{
   int i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
   int ibit,idim,k1,k2,n,nprev,nrem,ntot;
   float tempi,tempr;
   double theta,wi,wpi,wpr,wr,wtemp;

   ntot=1;
   for (idim=1;idim≤ndim;idim++)
      ntot *= nn[idim];
   nprev=1;
   for (idim=ndim;idim≥1;idim−−) {
      n=nn[idim];
      nrem=ntot/(n*nprev);
      ip1=nprev ≪ 1;
      ip2=ip1*n;
      ip3=ip2*nrem;
      i2rev=1;
      for (i2=1;i2≤ip2;i2+=ip1) {
         if (i2 < i2rev) {
            for (i1=i2;i1≤i2+ip1−2;i1+=2) {
               for (i3=i1;i3≤ip3;i3+=ip2) {
                  i3rev=i2rev+i3−i2;
                  SWAP(data[i3],data[i3rev]);
                  SWAP(data[i3+1],data[i3rev+1]);
               }
            }
         }
         ibit=ip2 ≫ 1;
         while (ibit ≥ ip1 && i2rev > ibit) {
            i2rev −= ibit;
            ibit ≫= 1;
         }
         i2rev += ibit;
      }
      ifp1=ip1;
      while (ifp1 < ip2) {
         ifp2=ifp1 ≪ 1;
         theta=isign*6.28318530717959/(ifp2/ip1);
         wtemp=sin(0.5*theta);
         wpr = −2.0*wtemp*wtemp;
         wpi=sin(theta);
         wr=1.0;
         wi=0.0;
         for (i3=1;i3≤ifp1;i3+=ip1) {
            for (i1=i3;i1≤i3+ip1−2;i1+=2) {
               for (i2=i1;i2≤ip3;i2+=ifp2) {
```

```
                k1=i2;
                k2=k1+ifp1;
                tempr=wr*data[k2]-wi*data[k2+1];
                tempi=wr*data[k2+1]+wi*data[k2];
                data[k2]=data[k1]-tempr;
                data[k2+1]=data[k1+1]-tempi;
                data[k1] += tempr;
                data[k1+1] += tempi;
              }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
          }
          ifp1=ifp2;
        }
        nprev *= n;
      }
    }

#undef SWAP

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

/******** COSMatrix.c creates the Cosine matrix. ***************

      Written By: Jim Goble
      Date:       1 July, 1991
      Version:    1.0

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


#include <math.h>
#include <stdio.h>
#define PI 3.14159265
main()
{
 FILE *Cfile,*ofp;   /* My storage file pointer */
 double temp,NN;
 double cos(),sqrt();
 int m, n, M, N, X, Y;
 printf("!!!  Input the desired number of rows:");
 scanf("%d", &N);
 printf("\n");
 /*printf("!!!  Input the desired number of cycles in Y:");
 scanf("%d", &Y);
 printf("\n");*/
```

```
 printf("!!!  Input the desired number of cycles in X:");
 scanf("%d", &X);
 printf("\n");

 M = N;
 NN = N;

 M = M - 1; /* increment variables */
 N = N - 1;

/* Open Cfile for writing.  Note there is no error checking! */

 Cfile = fopen("cos.dat", "w");

/*Compute the CMatrix */

 for (m = 0; m<= M; ++m) {
  temp = cos((2*X*m*PI)/NN);
  fprintf(Cfile,"%-8.7f\n", temp);
 } /* end of m for loop */
 fclose(Cfile);
} /* end of program */

/*************************************************************/
```

Appendix E. *Statistical Prediction Algorithm and Source Code*

This appendix contains an overview of the statistical prediction algorithm used to compare with the recurrent network prediction. Following the description of the algorithm is a listing of the C source code which empliments the statistical prediction algorithm.

E.1 Statistical Prediction Algorithm

Given an ergotic signal described by the function $x(t)$, the future value of $x(t)$ at time t_2 is given by

$$x'(t_2) = E[x_{t_2}] + \frac{cov(x_{t_1}, x_{t_2})}{var(x_{t_1})} (x(t_1) - E[x_{t_1}]) \tag{14}$$

where the expectation values (means) $E[x_{t_1}]$ and $E[x_{t_2}]$ are given by

$$E[x_{t_1}] = E[x_{t_2}] = \frac{1}{N} \sum_{1}^{N} x(n) \tag{15}$$

and the variance $var(x_{t_1})$ is given by

$$var(x_{t_1}) = E[x_{t_1}^2] - E^2[x_{t_1}]) \tag{16}$$

and the covariance $cov(x_{t_1}, x_{t_2})$ is given by

$$cov(x_{t_1}, x_{t_2}) = E[x_{t_1} x_{t_2}] - E^2[x_{t_1}]) \tag{17}$$

For the variance, the expectation value (mean) of $x_{t_1}^2$ is given by

$$E[x_{t_1}^2] = \frac{1}{N} \sum_{1}^{N} x^2(n) \tag{18}$$

and for the covariance, $E[x_{t_1}, x_{t_2}]$ is given by

$$E[x_{t_1}x_{t_2}] = \frac{1}{N}\sum_{1}^{N} x(n)x(n+k) \tag{19}$$

where k is some constant time in the future.

The measure of performance is the mean squared error between the predicted value and the actual value at some time in the future. This error is given by

$$error = e = E[(x'(t_2) - x(t_2))^2] \tag{20}$$

E.2 Source Code Listing

The following source code listing is a C emplimentation of the statistical prediction algorithm previously outlined. It requires two serarate files to be present in the same directory: "sp_defs.h", a declarations file for the main program, and "param_sp.dat", a parameter file for declaring variable arrays. The functions used from "NRUTIL.C" are listed in Appendix B.

```
/* SP.C ************************************************************

    Statistical Prediction Software.  This program performs
   the best linear prediction for any ergotic function.  The
   default input datafile name is "data.dat" but you can use
   any filename desired as long as it is passed to SP at the
   command line.  The results printed to the default
   display are self explanitory.
     Required input files:  sp_defs.h, and params_sp.dat
   Files created:  stat_results.dat, stat_des.dat
        stat_out.dat, and stat_error.dat

   date:    17 Oct 91

   written by:  Randall L. Lindsey, GEO-91D
*****************************************************************/

#include <stdio.h>
#include "macros.h"
#include <math.h>
```

```
#include "sp_defs.h"
#include <string.h>

void main(int argc, char *argv[])
{
 switch (argc) {
 case 1:
  datafile="data.dat";
  check_file();
  initialize();
  read_data();
  MEAN();
  MEAN_SQ();
  VAR();
  COV();
  compute_output();
  compute_error();
  print_results();
     break;

 case 2:
  datafile=argv[1];
  check_file();
  initialize();
  read_data();
  MEAN();
  MEAN_SQ();
  VAR();
  COV();
  compute_output();
  compute_error();
  print_results();
  break;

 case 3:
 default:
  printf("\nUsage:  sp [datafilename.dat] \n\n");
  break;
 }

} /* End MAIN() of SP.C */

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

void initialize()
{
 /* Read data from the input file "param_sp.dat" */
```

```
  printf("%s","Init...");
  ifp=fopen("param_sp.dat","r");
  fscanf(ifp,"%d %f %d %d",&epochs,&alpha,&seed,&look_ahead);
  fclose(ifp);

 /* Read data from the input file datafile (user specified) */

  ifp=fopen(datafile, "r");
  fscanf(ifp,"%d %d %d",&num_inputs,&num_outputs,&num_nodes);
  fscanf(ifp,"%d",&num_vectors);
  fclose(ifp);

  m = num_inputs + 1;   /* # of external inputs */
  nrows = n = num_nodes;   /* # of rows for weight matrix */
  ncols = m + num_nodes;   /* # of cols for weight matrix */

 /* Allocate memory for vectors and matrices */

  e=vector(0,nrows-1);      /* error vector       */
  z=vector(0,num_vectors+look_ahead);      /* input vector array  */
  y=vector(0,num_vectors);      /* output vector array  */
  d=vector(0,num_vectors+look_ahead);    /* desired output array */

 /* Initialize variables to zero */

  J[0]=J[1]=0.0;
  loopi(num_vectors)
    e[i] = y[i] = d[i] = z[i] = 0.;

  return;
}

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

void read_data()
{
  ifp=fopen(datafile, "r");
  fskip_line(ifp);
  loopi(num_vectors+look_ahead)
    fscanf(ifp,"%f  %f",&z[i],&d[i]);
  fclose(ifp);
  return;
}

/*****************************************************************/
```

```
void MEAN()
{
  float X=0.;
  loopi(num_vectors)
    X += z[i];
  mean = 1.0/(float)num_vectors * X;
  printf("%s = %f\n","Mean",mean);
  return ;
}

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

void MEAN_SQ()
{
  float X=0.;
  loopi(num_vectors)
    X += z[i]*z[i];
  mean_sq = 1.0/(float)num_vectors * X;
  printf("%s = %f\n","Mean Sq",mean_sq);
  return;
}

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

void VAR()
{
  var = mean_sq - mean*mean;
  printf("%s = %f\n","Var",var);
  return;
}

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

void COV()
{
  float X=0.;
  loopi(num_vectors)
    X += z[i]*z[i+look_ahead];
  cov = (1.0/(float)num_vectors * X) - (mean*mean);
  printf("%s = %f\n","Cov",cov);
  return ;
}

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

void compute_output()
{
```

```
  printf("%s\n","Output");
  loopi(num_vectors)
    y[i] = mean+(cov/var)*(z[i]-mean);
  return;
}

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

void compute_error()
{
  float X=0.;
  loopi(num_vectors){
    e[i] = z[i+look_ahead] - y[i];
    X += e[i] * e[i];
  }
  error = 1.0/(float)num_vectors * X;
  printf("%s = %f\n","Error",error);
  return ;
}

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

void print_results()
{
  printf("%s","Printing results...");
  ofp = fopen("stat_results.dat", "w");
  loopi(num_vectors)
    fprintf(ofp,"% f  % f  % f\n",y[i],d[i],e[i]);
  fclose(ofp);
  ofp = fopen("stat_out.dat", "w");
  loopi(num_vectors)
    fprintf(ofp,"% f\n",y[i]);
  fclose(ofp);
  ofp = fopen("stat_des.dat", "w");
  loopi(num_vectors)
    fprintf(ofp,"% f\n",d[i]);
  fclose(ofp);
  ofp = fopen("stat_error.dat", "w");
  loopi(num_vectors)
    fprintf(ofp,"% f\n",e[i]);
  fclose(ofp);
  printf("%s\n","Done.");
}

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

void check_file()  /* Written 10 Jul 91, RLL. */
```

```
{
  FILE *afp;

  afp = fopen(datafile,"r");
  if(afp == NULL) {
    /*strcpy(afile,"File not found");*/
    printf("\n%s %s\n",datafile,": File not found.");
    exit(0);
  }
  else fclose(afp);
  return;
}

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


/* SP_DEFS.H ***********************************************

     File containing function declarations and variable
     declarations for the main program called sp.c.

   date:  17 Oct 91

   written by:  Randall L. Lindsey
*****************************************************************/
float *vector();

FILE *ifp, *ofp, *ifp1, *ofp1;
int run=1;
char str[80], *datafile;
int nrows, ncols, i, j, k, l, m, n;
int epochs, a, b, t, look_ahead;
int num_inputs, num_outputs, num_nodes, num_vectors, seed;
float alpha, alpha1, J[2], sum, mean, mean_sq, var, cov, error;
float *e, *z, *y, *d;
void MEAN();
void MEAN_SQ();
void VAR();
void COV();
void initialize();
void read_data();
void compute_error();
void print_results();
void check_file();
void compute_output();

/*****************************************************************/
```

```
/** PARAM_SP.DAT ***************************************************/

200 3.0 153

/*****************************************************************/
```

Bibliography

1. Almeida, L. B. "A Learning Rule for Asynchronous Perceptrons With Feedback in a Combinatorial Environment." In *Proceedings of the IEEE First International Conference on Neural Networks, II*, pages 609–618, June 1987.
2. Fang, Yan and Terrence J. Sejnowski. "Faster Learning for Dynamic Recurrent Backpropagation," *Neural Computation*, 2:270–273 (1990).
3. Gaskill, Jack D. *Linear Systems, Fourier Transforms, and Optics*. New York: John Wiley and Sons, 1978.
4. Hecht-Nielsen, Robert. *Neurocomputing*. Reading, Massachusetts: Addison-Wesley Publishing Co., January 1991.
5. Hopfield, J. J. "Neural Networks as Physical Systems with Emergent Collective Computational Abilities." In *Proceedings of the National Academy of Sciences, 79*, pages 2554–2558, 1982.
6. Lapedes, A. and R. Farber. "A Self-Optimizing, Nonsymmetrical Neural Net for Content Addressable Memory and Pattern Recognition," *Physica D, 22*, pages 247–259 (1986).
7. Le, Capt Phung D. *Model-Based 3-D Recognition System Using Gabor Features and Neural Networks*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.
8. Mish, Fredrick C., editor. *Webster's Ninth New Collegiate Dictionary* (First digital Edition). Boston: Merriam-Webster Inc., and NeXT Computer, Inc., 1988.
9. Pearlmutter, B. A. "Learning State Space Trajectories In Recurrent Neural Networks," *Neural Computation, 1*:263–269 (1989).
10. Pineda, Fernando J. "Generalization of Back-Propagation to Recurrent Neural Networks," *Physical Review Letters, 59-19*, pages 2229–2232 (November 1987).
11. Pineda, Fernando J. "Recurrent Backpropagation and the Dynamical Approach to Adaptive Neural Computation," *Neural Computation, 1*:161–172 (1989).
12. Press, William H. and others. *Numerical Recipies in C*. Cambridge: The MIT Press, 1991.
13. Rogers, Steven K. and Matthew Kabrisky. *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition*. Washington: SPIE Optical Engineering Press, 1991.
14. Rohwer, Richard and Bruce Forrest. "Training Time-Dependence in Neural Networks." In *Proceedings of the IEEE First International Conference on Neural Networks, II*, pages 701–708, June 1987.

15. Rosenblatt, F. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Washington: Spartan Books, 1959.

16. Ruck, Dennis W. *Characterization of Multilayer Perceptrons and their Application to Multisensor Automatic Target Detection*. PhD dissertation, Air Force Institute of Technology, Wright-Patterson AFB, OH, December 1990 (AFIT/DS/ENG/90-2).

17. Ruck, Dennis W., et al. "Feature Selection Using a Multilayer Perceptron," *The Journal of Neural Network Computing*, 2(2) (Fall 1990).

18. Rumelhart, David E., et al. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Volume 1. Cambridge: The MIT Press, 1988.

19. Stright, James R. *A Neural Network Implementation of Chaotic Time Series Prediction*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988.

20. Switzer, Capt Shane R. *Frequency Domain Speech Coding*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

21. Williams, Ronald J. and Jing Peng. "An Efficient Gradient-Based Algorithm for On-line Training of Recurrent Network Trajectories," *Neural computation*, 2:490–501 (1990).

22. Williams, Ronald J. and David Zipser. "Experimental Analysis of the Real-time Recurrent Learning Algorithm," *Connection Science*, *1*(1):87–111 (1989).

23. Williams, Ronald J. and David Zipser. "A Learning Algorithm for Continually Running Fully Recurrent Neural Networks," *Neural Computation*, *1*:270–280 (1989).

24. Zipser, David. "A Subgrouping Strategy that Reduces Complexity and Speeds Up Learning in Recurrent Networks," *Neural Computation*, *1*:552–558 (1990).