AFIT/ENG/GE/94J-01

SUBGROUPED REAL TIME RECURRENT LEARNING NEURAL NETWORKS

THESIS

Jeffrey S. Dean, B.S.E.E.

AFIT/ENG/GE/94J-01

AFIT/ENG/GE/94J-01

SUBGROUPED REAL TIME RECURRENT LEARNING 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

Jeffrey S. Dean, B.S.E.E.

May 1994

Approved for public release, distribution unlimited

Acknowledgements

When I began working towards a Master's Degree at AFIT as a part-time student, I had only a vague idea of what my thesis would be about, and in the process of four years of taking my classes one at a time, that idea changed often. It was through several conversations with Professor Steve Rogers that I began to see the best of all possible choices. Not only are neural networks a "hot" topic these days, but they afforded me the opportunity to combine my background in Biology (B.A. in Biology from St. Louis University) with my engineering training. The more I learned about how living things process information, and how we were beginning to be able to emulate this ability, the more fascinating the subject became. I wish to thank Professor Rogers and Dr Matthew Kabrisky for helping me to make this connection, and for the knowledge we seem to soak up so easily in their presence. I also wish to thank Capt Dennis Ruck for his help with the RTRL algorithm, Maj Gregg Gunsch for keeping me pointed in the right direction. Last, but certainly not least, I wish to thank my wife and family for putting up with me over the past five years as I slowly earned my Master's degree. Whenever I became too mired in keeping up the job and AFIT, my wife Marla always pulled me out of it. There were many times when my children had to be AFIT orphans, and I thank them for their continued love and support.

Jeffrey S. Dean

Table of Contents

List of Figures

Figure

Abstract

A subgrouped version of the Real Time Recurrent Learning (RTRL) network was written in C, and its capabilities were evaluated. Although the RTRL net architecturally consists of one layer of neurons it successfully learns the XOR problem, and can be trained to perform time dependent functions such as emulating a digital low pass filter, and internalizing a state model of a data sequence. The net was tested as a predictor, to evaluate it's ability to predict the future value of a chaotic signal based on past behavior. While the net was not able to predict a chaotic signal's future output, it tracked the signal closely. The net was also tested as a classifier for time varying phenomena; for the differentiation of five classes of vehicle images based on features extracted from the visual information. The net achieved a 99.2% accuracy in recognizing the five vehicle classes. Recognition was based on the sequences of vector quantized codewords which represented feature changes caused by shifting the vehicle image aspect over time.

The various operating parameters of the subgrouped recurrent net program (initial learning rate, momentum, minimum allowed sigmoidal derivative, teacher forced learning, weight update error threshold and continuity of recurrence between training epochs) were tested for their impact in learning performance, as applied to phoneme group classification and a low pass Butterworth filter emulation. The behavior of the subgrouped RTRL net was compared to the RTRL net described in Capt Randall Lindsey's AFIT Master's thesis(7). Varying the net operating parameters demonstrated how gains in network error reduction could be obtained, and the subgrouped RTRL network performance proved close to the RTRL algorithm in accuracy while reducing the time required for updating network weights during training for a multiple output (classification) problem.

A SUBGROUPED REAL TIME RECURRENT LEARNING NEURAL NETWORK

I. Introduction

Neural networks have been receiving a tremendous amount of interest lately, not only from the engineers and researchers who are applying them to solve problems, but from the non-technical general population as well. They are often likened to the human brain, learning from experience to solve general problems. While an intriguing analogy, any attempt to imply that neural networks work in the same way as a human brain is misleading. Neural networks are computer algorithms, many forms of which were inspired by the apparent method in which neurons process information in biological systems. New variations of neural networks are being generated continuously, and the best type of neural net to apply depends on the characteristics of the problem being solved.

Many of the problems being attacked by neural networks are time dependent, i.e. the pattern learned by the network varies over time, and each state of the output is in some way dependent on information processed prior to that point. This makes it essential to know what happened in the past to correctly process the current data. To solve such tasks with neural nets requires some method of capturing temporal information. Recurrent neural networks perform this feat by feeding back information from the hidden and/or output nodes back into the network inputs. This allows the network to see the current data as well as a processed version of prior input data.

The addition of temporal information may make a recurrent network better at solving problems such as predicting commodity prices, identifying moving targets or identifying the different sounds, called phonemes, in human speech.

1.1 Problem

The Real Time Recurrent Learning (RTRL) network is a recurrent neural net that has been proven to be able to learn time dependent functions such as tracking analog signals, imitating a digital filter and recognizing sequences (17)(7). One well known (20) limitation of the RTRL algorithm however is the level of computer processing required for updating the weights, which is on the order of O(neurons^4). This makes large, multiple network output problems expensive computationally to train, and in some cases impractical. The goal of this thesis was to determine the behavior and performance of the *subgrouped* RTRL network described by Zipser (20). This variation of the RTRL algorithm reduces the computational requirements for training the network for multiple output problems requiring larger numbers of neurons.

The problem faced in this thesis was to quantify the behavior and performance of the subgrouped RTRL network, and to apply it to problems where the characteristics of the net will be beneficial. Because the subgrouped RTRL network is a time dependent neural network, it was applied to two problems with inherent time dependencies within the data:

A. Predicting the daily opening values of the pound in the London Exchange based on past performance.

B. The problem of classifying images based on sequences of vector quantized data, representing aspect or point of view changes in the observation of 5 different vehicles over time.

1.2 Background

With the myriad symposia, conferences, and publications currently devoted to neural nets, it is often difficult to maintain a current understanding of the "state of the art" in neural networks. Not only are new forms of networks continually being developed, but the more established neural networks (Cybenko, feedforward, Hopfield, Adaptive Resonance Theory, etc.) are continuously being modified, tweaked and improved upon, creating a multitude of related offspring. This thesis will focus on those networks that specifically incorporate time as part of the processing of information, and particularly on the subgrouped Real Time Recurrent Learning (RTRL) network.

1.3 Scope

The scope of this thesis is to characterize the behavior of the subgrouped RTRL network, as applied to the problems examined. This includes its application to the prediction of the opening value of the pound on the London Exchange, and the vehicle identification problem based on sequences of feature vectors(3) as the image aspect changes over time. The subgrouped RTRL network is a modified version of Capt Randall Lindsey's thesis program (7), which is based on the RTRL algorithm(17)(20). Comparisons in performance of the RTRL and subgrouped RTRL nets are also made, to determine how subgrouping impacts the training time and accuracy to the network.

1.4 Approach

The differences between the performance of the RTRL and subgrouped RTRL networks will be examined by performing the several of the demonstration tasks performed in Lindsey's thesis. This will determine whether the subgrouped RTRL network has the same functionality as Lindsey's RTRL code.

The network will also be evaluated as a predictor and as a classifier. The ability to predict will be examined by training the network on historical data derived from one

year's worth of opening values for the pound on the London Exchange, with the desired output being the opening value of the next day. After training, the net will be tested using opening value data from a different year.

The network's ability to classify will be evaluated by applying the subgrouped RTRL network to the problem of image identification. The network will be trained to differentiate between the images of five different vehicles, based on sequences of vector quantized codewords which encode changes in aspect as the viewing angle on the vehicles changes over time. The 4000 sequences in the data source file (800 sequences per vehicle, five vehicles) will be placed in random order and divided, using the first 90% of the sequences for training the network, and using the other 10% to test the accuracy of the network after training.

Chapter II provides background information on neural networks, and on time dependent neural networks in particular. It also discusses the source of the Pound monetary exchange rate values used to test the net's ability to predict, and the preprocessing of the data used for the vehicle classification problem. Chapter III delves into the algorithms used by the RTRL and subgrouped RTRL networks, discusses several operating parameters to the net to that can be changed to enhance performance, and reviews the test methodology used to characterize the capabilities of the subgrouped RTRL network. In Chapter IV, the testing results are examined, and in Chapter V conclusions and recommendations are presented.

II. Literature Review

2.1 Introduction

The purpose of this literature review is to synopsize the current state of time dependent neural networks, with particular attention paid to recurrent neural networks.

Neural networks represent man's attempts to learn from nature's multi-billion year experiment with life, in which the more effective and advantageous methods of living in a potentially hostile world are passed on and improved through the generations of living things. Because of nature's head start on us in developing sophisticated methods of coping with the environment, we are only now developing systems with the capacity that insects take for granted, i.e. pattern recognition, feature extraction, and autonomous travel.

Recurrent neural networks are a subset of the many varieties of neural nets, and have the added ability of incorporating time dependency into the evaluation of data. As many phenomena currently being evaluated with neural networks are time varying (speech, visual processing), this property may be essential to creating systems that may understand the spoken word, or interpret it's visual environment.

This section contains an overview of neural network theory, leading into a discussion of the various neural networks that incorporate changes over time into their training and function. The focus will be on Time Delay Neural Networks (TDNN), backpropagation through time (BPTT), real-time recurrent learning (RTRL), and sub-grouped RTRL networks.

2.2 *Background*

Neural networks are algorithms often based on the observed collective behavior of neurons in biological systems. In living organisms possessing a nervous system, neurons interconnect with each other as well as with sensory organs and muscles. The strength of the signals transferred to a neuron depends on the number of synaptic connections from other neurons, the activity (nerve depolarizations per second) of a stimulating neuron, the added stimulation or inhibition provided by other neurons, and how fast the neurotransmitters being produced at the synapses are broken down and reabsorbed. All these factors can be considered as weighted inputs which influence whether the neuron receiving the stimulus will fire, and how fast it will fire. This is modeled in neural networks by attributing weights to the interconnections in the network, and modifying the value of the weights in order to train the network to perform a function.

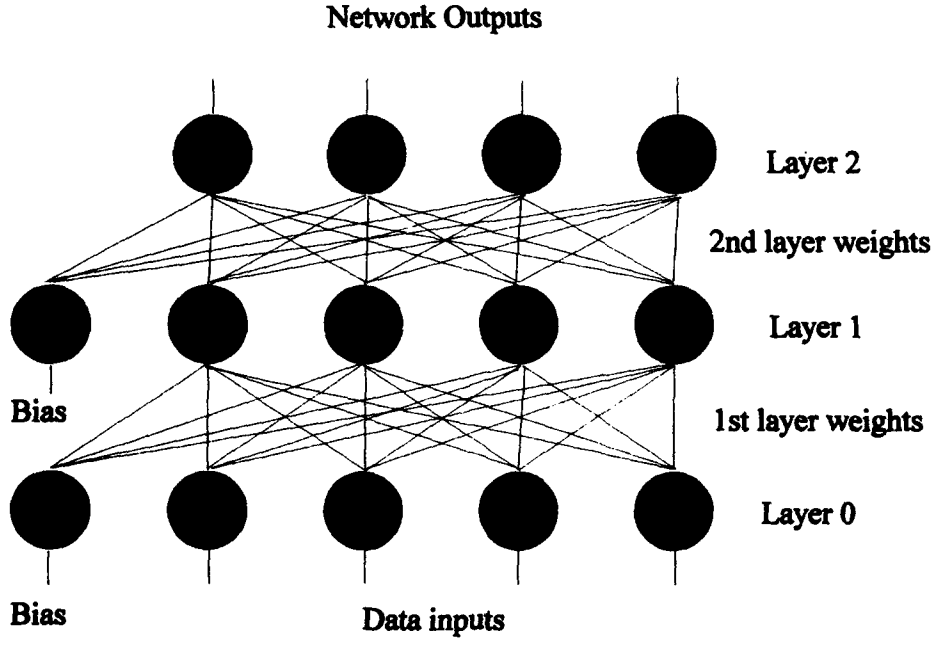


Figure 1: A two layer multilayer perceptron backpropagation network

In the standard multilayer perceptron network, the "neurons" are arranged in layers. Figure 1 shows a two layer network which will be discussed in the following paragraphs. Data feeds into the lowest layer, and is often represented as layer zero of the network. In multilayer networks, each level below the output layer provides inputs to the next higher layer. Each neuron in the network multiplies each of its inputs by a weight associated with that input, and sums the products together. This sum for a neuron i receiving j inputs can be described by:

$$s_i = \sum_{j=1}^{n} w_{ij} x_j + b_j \qquad (1)$$

The weight w_{ij} is a member of a matrix, with i ranging from one to the number of neurons in the layer containing neuron i, and j ranging from one to the number of inputs from the layer below. The input x_j represents either the outputs of the preceding layer or, if x_j lies on the lowest layer, the data being fed into the net, while b_j is a bias added to the inputs. If the neuron i is linear, s_i is the output of the neuron. If the neuron has a non-linear output function however, the sum is fed into the non-linear function (sigmoid, tanh, hard limiter or threshold) to produce the final output.

Training of the network is accomplished by adjusting the weights incrementally in a way that reduces the error between the output of the neuron and its desired output, which for the top layer of the network is shown as

$$e_i = d_i - y_i \qquad (2)$$

where y_i is the output of neuron i, and d_i is the neuron's desired output. If the neuron has a linear output, the error in the output of neuron i caused by weight w_{ij} depends on input x_j multiplied by the weight w_{ij} . Changing the weights to reduce the error can be performed by a simple formula

$$w^{+}_{ij} = w^{-}_{ij} - \eta e_i x_j \qquad (3)$$

where η is the learning constant for updating the weights, and w^- and w^+ refer to the weight prior to and after updating, respectively.

If the output of the neuron is non-linear, the weight update is a little more complicated. In the case of a sigmoidal output function, one of the most commonly used non-linear functions, the summed inputs of the neuron are processed by the formula

$$f(s_i) = 1/(1+e^{-s_i}) \tag{4}$$

In this case, the change in the error for that neuron for the weight being updated ($\delta e_i/\delta w_{ij}$) depends on the input that was multiplied by the weight and the derivative of the non-linear function. For the sigmoid function, the derivative is

$$f(s_i)(1-f(s_i)) \tag{5}$$

leading to a weight update formula of

$$w^+_{ij} = w^-_{ij} - \eta e_i f(s_i)(1-f(s_i))x_j \tag{6}$$

If the network has a layer of neurons below the output layer (usually called a hidden layer), there is no set desired output for these neurons to train on. Instead, the error generated by these neurons must be inferred by their net effect on the error of the output layer neurons. This carrying of the errors produced at the output of the network back to the hidden layers is the origin of the term backpropagation. For a neuron j in the hidden layer this error depends on the weights between neuron j and the output neurons, and on the derivative of the sigmoidal function used by the output neurons. This can be summarized by

$$\sum_{i=1}^{n} w_{ij}\, e_i\, f(s_i)(1-f(s_i)) \tag{7}$$

Using this term for the change in the output error generated by the output of neuron j, and with the same dependencies on the sigmoidal function and the inputs into neuron j for updating the weights as was seen in the output level, we can update the hidden layer weights using

$$w^{+}_{jl} = w^{-}_{jl} - \eta f(s_j)(1-f(s_j))x_l \sum_{i=1}^{n} w_{ij}\, e_i f(s_i)(1-f(s_i)) \qquad (8)$$

where w_{jl} represents the weight matrix used to weight the inputs from the next lower level. For a network with only two layers of neurons, x_l is one of the data inputs being fed into the network.

By updating the weights in the network incrementally over multiple (often thousands) of epochs in which the input data is passed through the network each epoch, the weights eventually reach a point at which the error has reached a minimum. This minimum may be the lowest possible error that can be achieved, or it may be a "local minimum" in which the net has become caught. Because changing the weights incrementally to travel between a local minimum and the global minimum would raise the output error temporarily in the process, the learning algorithm described above may not be able to reach the lowest possible output error.

The preceding paragraphs provide a top level, non-mathematically intensive description of how a standard backpropagation neural net operates. For the interested mathematically inclined reader, many excellent texts provide a detailed derivation of the backpropagation algorithm (12).

2.3 Scope of Literature Review

Because this thesis is based on the use of a time dependent neural network, specifically the subgrouped RTRL algorithm, this review will focus on those types of neural networks that are designed to incorporate time as a dimension in the training and function of the network. There are many forms of networks that use time in some manner, with variations and entirely new architectures being introduced regularly. Therefore, the broad classes of the currently well known time based neural networks will be discussed. A brief description of the derivation of image features used for the vehicle classification problem is also provided.

2.4 *Time Delay Neural Networks*

The element of time can be incorporated into the training of neural networks in several ways. The inputs into the network may include more than one "frame" of the training data, which is shifted through as the net is trained (Figure 2). The Time Delay Neural Network (TDNN) (Figure 3) operates on the same principle, where each input is split N times, with each of the N branches delayed by a different increment in time. This widens the window that the net "sees" of the data, to incorporate N time slices of the data stream. Waibel(18)(19) has used this type of network with some success for the identification of phonemes in Japanese.

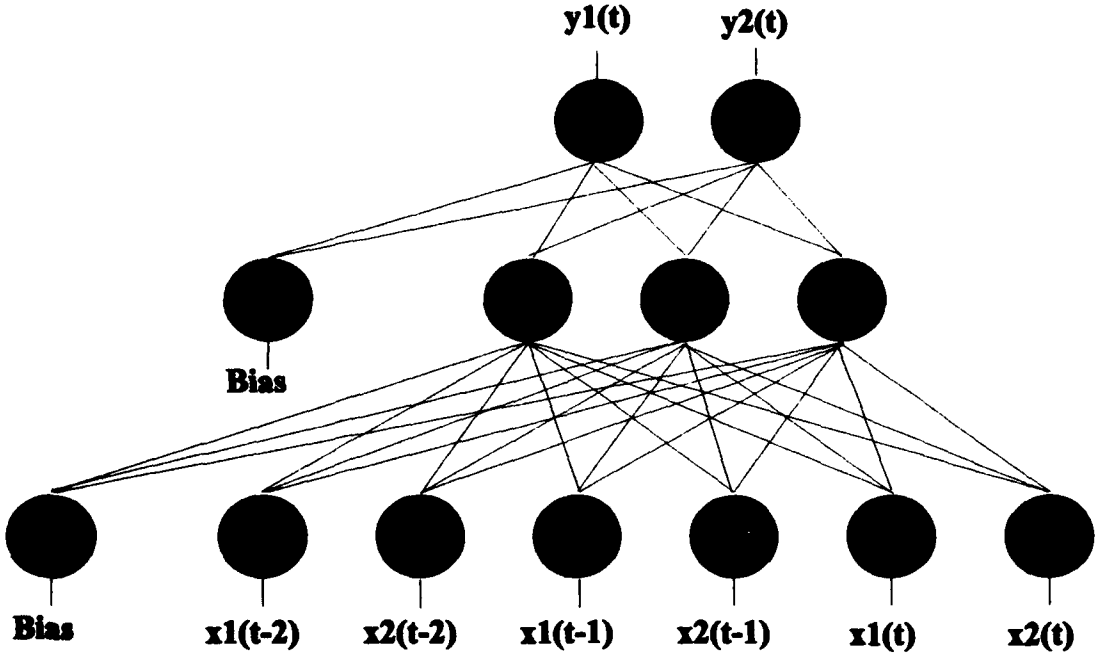


Figure 2: Input data is shifted along inputs to the net. In this example, the net sees three time samples of two inputs.

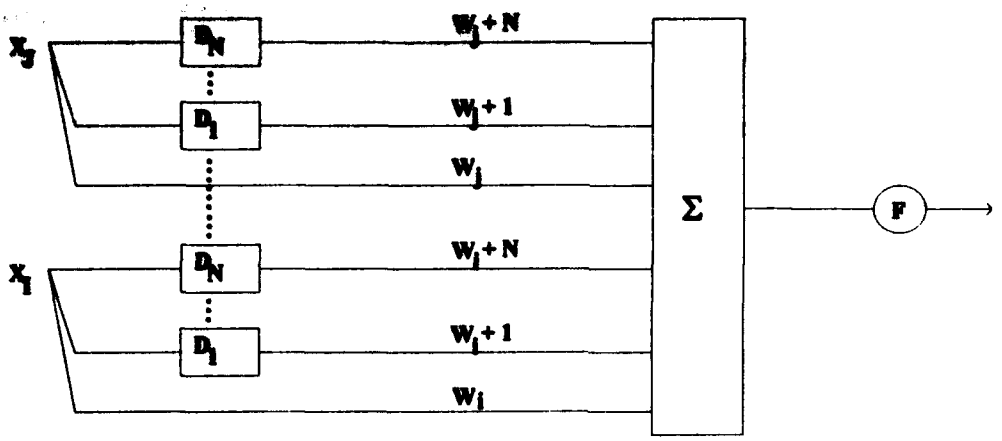


Figure 3: Basic neuron in Time Delay Neural Network. Each input is split N+1 times, with each version of the input delayed a different time increment and multiplied by a weight.

2.5 Recurrent Network Variations

Recurrency in a neural network basically involves the feeding back of the outputs of neurons in the network to other neurons on the same layer or at lower levels. Jordan(6) proposed a network that operated like a standard two layer backpropagation network, but fed back the outputs of the network as inputs, allowing the net to "see" what was produced during the last iteration (Figure 4). The recurrent output values were fed into the hidden layer neurons, as well as having each state unit neuron feeding back to itself, multiplied by an attenuation factor. Elman(2) described a variation on this concept, in which the output of the hidden nodes is fed back as net inputs (Figure 5). These recurrent architectures are straightforward, in that no changes to the standard backpropagation algorithm is required. The recurrent values are treated as inputs, and the net performs a gradient descent to minimize the error as it trains.

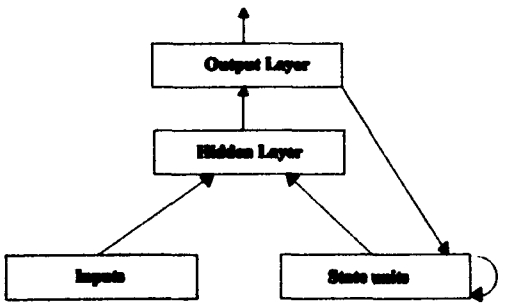


Figure 4: Simple recurrent net, where the outputs are fed back into the net with each iteration

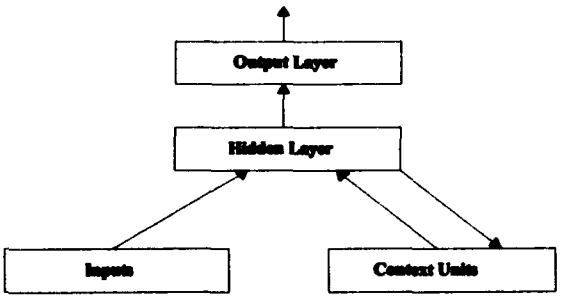


Figure 5: Simple recurrent net, where the activation values of the hidden layer are fed back into the net with each iteration

Rumelhart(15) proposed a different tack in approaching the treatment of time. In his recurrent network, the network is treated as a feedforward network that grows one layer with each iteration. This algorithm is known as back-propagation through time (BPTT), and while it does solve time-dependent problems it suffers from computer resource limitations, as the net grows larger with larger input sequences. Rohwer and Forest(13) modified this approach by creating multiple copies of the starting network, with each copy representing a time step in the training sequence.

Pineda(10) generalized Rumelhart's(15) learning rule, while eliminating the requirement to unfold or duplicate the network for each time step. The net, similar in some ways to the Hopfield network, is designed to adjust the weights in order to produce a fixed point (corresponding to a memory in a Hopfield net) when an input x_i is presented to the net in an initial state x_i. Unlike the Hopfield net, the weights are adjusted to minimize the error of the system during training.

Pineda (10) later stated that gradient descent cannot create new fixed points, only move existing ones. To create new fixed points requires "teacher forcing", which constrains the degrees of freedom in the network during training, and releases them during recall. He also states that there is no guarantee that after the clamped degrees of freedom are released that the system will be stable on those fixed points, and that the fixed points generated by the clamping may become "repellers" rather than "attractors."

Pineda's(10) algorithm for training recurrent networks was adapted and generalized by Pearlmutter(9) to minimize the net error as a function of the temporal trajectory of the states of the network. This new algorithm trained slowly and occasionally became unstable, and was modified by Fang and Sejnowski(4) to overcome these obstacles.

2.6 *Real-Time Recurrent Learning (RTRL)*

Another variation in the recurrent network taxonomy is the real-time recurrent learning (RTRL) network proposed by Williams and Zipser(17) (Figure 6). It also minimizes the net error along a temporal trajectory using gradient descent, and can be used to recognize temporal sequences. Unlike the BPTT algorithm and many of its derivatives however, the network does not grow over long training sequences. The RTRL network does suffer from large memory and processing requirements, as the algorithm requires $O(n^4)$ computations per time step for n neurons. Because of this, this algorithm can be unsuitable for any problem that requires a combination of multiple (>10) inputs, multiple (>3) outputs and associated hidden units.

Because the net processes information by passing the output of the neurons back as inputs at the next point in time, the training output values provided to the net must be delayed one or more time steps as compared to the corresponding network training inputs.

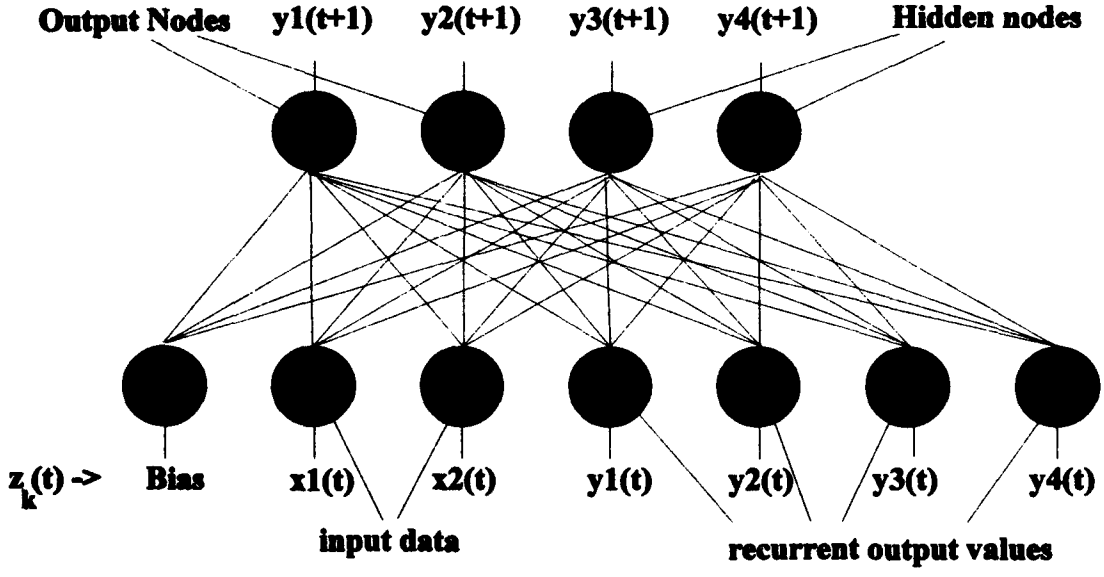


Figure 6: Basic RTRL architecture, with two outputs, two hidden nodes, and two inputs.

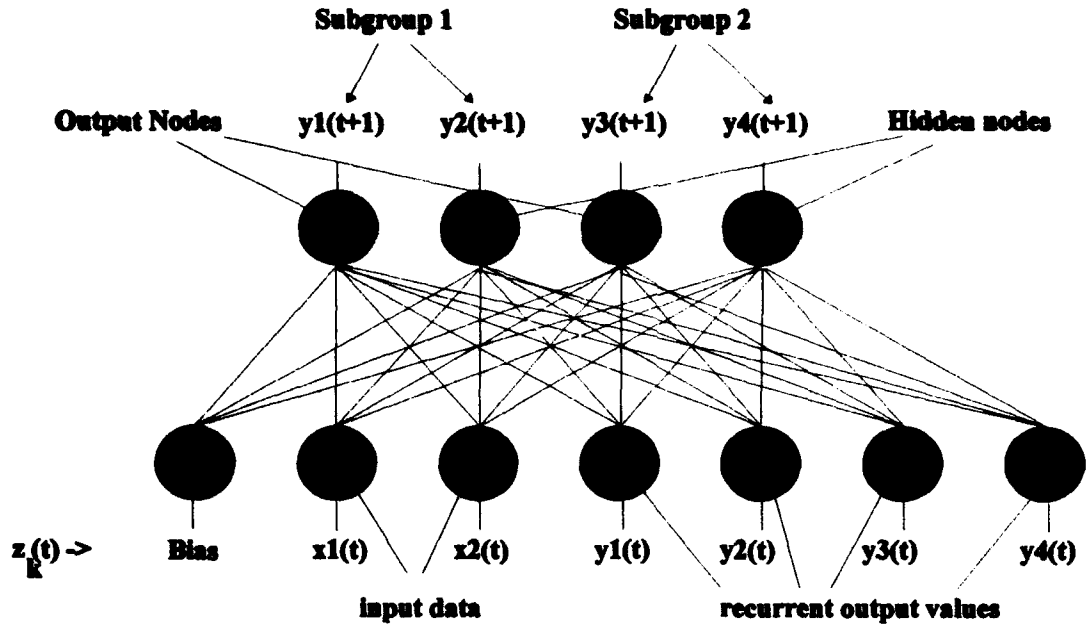


Figure 7: The subgrouped RTRL architecture, as implemented for this thesis. Each output is paired with one or more hidden nodes. Note the connectivity between the nodes is the same as in the basic RTRL architecture.

2.7 Subgrouped RTRL

To address the exponential growth in computational requirements of RTRL, Zipser(20) proposed a method of breaking the updating of the weights into subgroups (Figure 7). This method can reduce the computational complexity of the algorithm from $O(wn^2)$ to $O(w)$, where w represents the size of the weight matrix and n equals the number of neurons. The connectivity within the network is unchanged, but the updating of each weight depends on only a subset of the error generated by the recurrent neuronal outputs. For g subgroups the weight updating algorithm is g^2 times faster, although each subgroup now has less of the temporal "memory" than was found in the original algorithm. Zipser(20) states that this can be compensated for by using more hidden units, while still operating at a much faster processing rate.

Like the RTRL algorithm, network training outputs must be delayed by one or more time steps from the corresponding network inputs. The explanation of the RTRL

algorithms, and how subgrouping speeds up the network, is discussed in section 3.2. It is the subgrouped RTRL algorithm that will be the focus of this thesis.

2.8 London Exchange Opening Quotes

There are many examples in this world of data whose changes over time appear on the surface to be random or chaotic, but are dependent on some underlying mechanism that drives (or influences) the path the data takes. One example of this would be the amplitude of a vocal signal, dictated by the mechanics of the vocal chords and the phoneme being uttered at the moment. Another possible example would be the movement of a pilot's head in XYZ space during a mission, which would be influenced by the voluntary movements (looking at the Heads Up Device) and the inertial forces generated by aircraft maneuvering. The ability to predict the path of a signal based on past behavior could be very beneficial, and would depend on the predictor's ability to internalize and emulate the mechanisms or forces that drive the signal to change. For this thesis, the opening quotes for the value of the pound on the London Exchange are used to study the subgrouped RTRL net's ability to perform this function.

2.9 Vector Quantized Image Sequences

The ability to visually recognize objects is one that we take for granted, unless we try to duplicate this ability in a machine. Generally, this is performed (or attempted) by extracting key visual features that are characteristic for the object being identified. The data used for this thesis was derived from CAD generated 3-D images of an M-60 tank, an M35 truck, a BTR60 armored personnel carrier, a T62 tank, and an M2 infantry fighting vehicle. Each image was viewed from multiple (592) different angles around and above the vehicle representations, and the data was processed and vector quantized(3) to produce 64 codewords. Each codeword (0 - 63) represents the visual information of areas of similar aspect or characteristic view. Codewords may be associated with one or more

of the vehicles; the key information is contained in the *sequences* of codewords, representing a changing image aspect over time.

2.10 Summary

Recurrent neural networks have grown in complexity from a basic feeding back of the output error of neurons at higher levels(2)(6) to algorithms that specifically incorporate the function of time into the weight updates. Because of this, these algorithms are uniquely capable of following the "trajectory" of the data through the time steps, allowing the network to predict what can be expected to occur next and respond accordingly. This added dimension of time expands the observed behavior of neural nets in generating a probability function as an output, in that the preceding time steps add to the network's ability to generate the most likely output.

The subgrouped RTRL algorithm is a flexible, time-dependent method of predicting what the most likely output should be, given the current inputs fed into the net at this time *and* the inputs that were fed into the net in the past. As such, its abilities and limitations need to be evaluated and explored. The full description of this algorithm, and the tests performed in this thesis to evaluate its effectiveness, are documented in Chapter III.

III. Methodology

3.1 Introduction

Chapter II provided an overview of how the standard multilayer network with backpropagation learning operates, as well as a review of the more prominent networks that are designed for the processing of temporally-dependent data. The real-time recurrent learning (RTRL) neural network and the subgrouped RTRL were highlighted due to their importance to this thesis. This thesis involves the modification of the RTRL C code written by Capt Lindsey(7) into the subgrouped RTRL algorithm, and the enhancement of the performance and learning effectiveness through several modifications. The utility of the algorithm is demonstrated via its application to the prediction of the value of the English pound based on the opening values at the London Exchange, and the identification of vehicle images based on image features as the viewing aspect changes with time.

This chapter covers the development, modifications and testing of the subgrouped RTRL program. The algorithm for the subgrouped RTRL, and how it differs from the basic RTRL algorithm, is described and discussed. Also, the training and testing methodology is reviewed.

3.2 Subgrouped RTRL Algorithm

Like the basic RTRL algorithm, the subgrouped RTRL is an error gradient following algorithm for a completely recurrent network. The subgrouped RTRL algorithm is structurally the same as the basic RTRL algorithm; the same connectivity exists between the nodes as in the RTRL. The main difference lies in the extent to which each node influences the weight updates of the network.

In the implementation of the subgrouped RTRL for this thesis, some restrictions into the algorithm have been incorporated to simplify the design. Both Lindsey's(7)

original RTRL code and the modified subgrouped RTRL allow the user to specify the number of output nodes, input nodes and hidden nodes. The user's selection of the number of hidden nodes however is changed, if required, to make the number of hidden nodes an integer multiple of the number of output nodes selected. Each output node is then grouped with an equal number of hidden nodes. As in the basic RTRL algorithm, the linear and/or sigmoidal outputs of the output and hidden nodes are fed back into base of the net, comprising part of the input for the next iteration.

The subgrouped RTRL algorithm was proposed by Zipser(20) in response to observations that the RTRL algorithm required a great deal of computation to train. This is due to the $O(n^4)$ complexity of each time step, with n representing the total number of output and hidden nodes. This thesis will review the subgrouped RTRL algorithm, and demonstrate where it deviates from the basic RTRL algorithm. Terms used in this derivation will be consistent with those used in Zipser's(20) article and Lindsey's(7) thesis. Specific portions of the discussion are attributed to the subroutines in the C code in Appendix B, to help the reader associate the algorithm to its implementation.

Basic terms:

The network (Figures 6 & 7) consists of n neural node units and has m external inputs (the first of which is a bias of 1). At time t,

the output of the kth neuron is represented by $y_k(t)$, where k ranges from 0 to n -1.

the summed activation value of neuron k at time t is $s_k(t)$

the jth external input into the net is $x_j(t)$, where j ranges from 0 to m - 1.

the $m + n$ net inputs comprise the input vector as time t, $z_j(t)$, where j ranges from 0 to $m + n$ - 1. This is shown by:

$$z_j(t) = \begin{cases} x_j(t) & \text{if } j \in \mathbf{I} \\ y_j(t) & \text{if } j \in \mathbf{U} \end{cases} \qquad (9)$$

where U identifies the subset of the j indices in z_j derived from the n network outputs of the prior iteration, while I identifies the subset of j indices in z_j in which the jth member is one of the m external inputs.

the error measured at neuron k is represented by $e_k(t)$.

the Kronecker delta function, δ_{ik}, equals 1 if $i = k$, and is 0 otherwise.

the non-linear sigmoidal function at neuron k is shown as f_k

the p_{ij}^k matrix represents $\delta y_k(t+1)/\delta w_{ij}$, where in the original RTRL network, i ranges from 0 to n -1, j ranges from 0 to $m + n$ -1, and k ranges from 0 to n-1. In the subgrouped RTRL net, i ranges from 0 to g - 1, where g is the size of the net subgroups, j ranges from 0 to $m + n$ - 1, and k ranges from 0 to n - 1.

As was covered in the discussion of the basic backprop net (Chapter II) there is a weight matrix w_{ij}, where i is the index of one of the n neurons, and j refers to one of $m+n$ inputs.

Subroutine Compute_Error:

This routine calculates the error at the net outputs, based on the net's *prediction* of what the output should be, which was calculated during the prior iteration. The error is found by taking the difference between the linear or sigmoidal output of those nodes designated as "output" nodes, and the desired output of the network. The error at each output node k is defined as

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T \\ 0 & \text{otherwise} \end{cases} \qquad (10)$$

where T represents the subset of neurons that produce the net outputs.

In the original RTRL code (Figure 6), the first k nodes were output nodes, while the remaining n - k nodes were the hidden nodes. For this implementation of the subgrouped RTRL (Figure 7), the output nodes are every ith neuron, where i = (n / number of net outputs).

The total mean squared network error is then calculated as

$$J_{total}(t) = \sum_t (1/2) \sum_{k \in J} [e_k(t)]^2 \qquad (11)$$

Subroutine Reset_Delw_S:

This subroutine multiplies the delta weight matrix with a momentum term after each iteration of data is processed by the net, allowing the net to use momentum while training. The rationale for the addition of the momentum term is discussed in section 3.3.2.

Subroutine Propagate:

The $y_k(t\text{-}1)$ outputs of the n neurons that were computed during the last iteration are incorporated in the input vector $z(t)$. The nodal activation, or the weighted sum of the inputs for each node (s_k), is calculated as

$$s_k(t) = \sum_{l \in U \cup I} w_{kl}\ z_l\ (t) \tag{12}$$

In other words, each neuron sums all of its inputs multiplied by their respective weights to form the activation value for that neuron at time t.

Subroutine Compute_Output:

The output for the following iteration is calculated next. This is expressed by

$$y_k\ (t+1) = f_k\ (s_k\ (t)) \tag{13}$$

where f_k is the sigmoidal function for the hidden nodes, and can be sigmoidal or linear for the output nodes. This $y_k(t+1)$ term is the network's predicted value of what the desired value will be next iteration.

Subroutine Update:

The updating of the weights in this algorithm is the most complex and computationally intensive portion of the RTRL algorithm. It was due to the computational requirements of this function that the subgrouped RTRL algorithm was

proposed by Zipser(20), and utilized for this thesis. To understand this, we must look at the effects of using recurrence in the network.

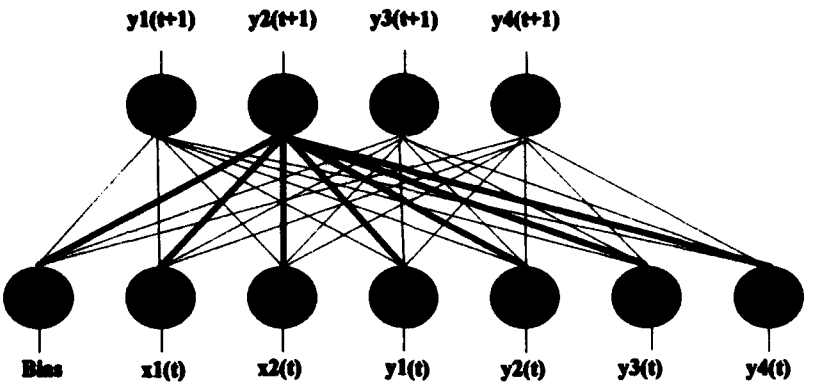


Figure 8: The output of neuron y2 at time *t*+1 is dependent on the highlighted weight connections and their inputs.

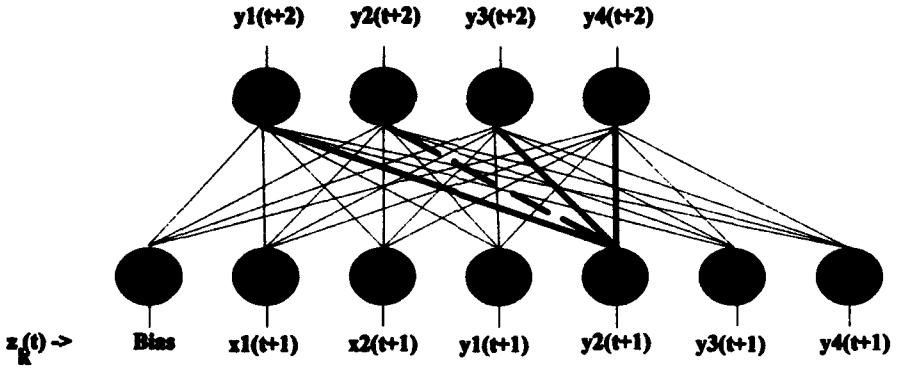


Figure 9: At the next iteration, the output of neuron y2 has been fed back into the network as an input, thereby affecting the output (and error) of each of the neurons at *t*+2. Note that neuron y2 can affect its own output (dashed line) during the next iteration.

In order to calculate the update to the weight w_{ij} for the next iteration using the RTRL algorithm, we must look at the error in the net caused by that weight. Weight w_{ij} affects the output of neuron i at time t (Figure 8). Since the output of the neuron i is fed back into the net along with whatever error it may contain, w_{ij} impacts the error in the next iteration of all the neurons (Figure 9).

The relationship between the energy level in the network and the network weights is represented by:

$$\frac{\delta J(t)}{\delta w_{ij}} = \sum_{k \in U} e_k(t) \frac{\delta y_k(t)}{\delta w_{ij}} \tag{14}$$

Because this is a recurrent network, a change in w_{ij} at time t affects the output and error of neuron k at time $t+1$. For the RTRL network, this is expressed by

$$\frac{\delta y_k(t+1)}{\delta w_{ij}} = f_k'(t)\left[\sum_{l \in U} w_{kl}\, p_{ij}^l + \delta_{ik}\, z_j(t)\right] \tag{15}$$

where $k \in \mathbf{U}$, $i \in \mathbf{U}$, and $j \in \mathbf{U} \cup \mathbf{I}$. The term p_{ij}^l represents the effect that a change in weight w_{ij} would have on the output of neuron l at a following iteration. Since neuron l is then fed back into the network and becomes an input to neuron k at time $t+1$, the $\sum_l w_{kl} p_{ij}^l$ term is a summation of each of the weights associated with the recurrent inputs to neuron k, times the changes in the recurrent inputs that were caused by weight w_{ij}. In other words, this term sums the indirect effect that weight w_{ij} has on the output of neuron k from changing the output of neuron l.

If neuron i and neuron k are the same (separated by time), the effect of a change in weight w_{ij} on the output of neuron k can be expressed by the added term

$$\delta_{ik} z_j(t) \tag{16}$$

The δ_{ik} term is the Kronecker delta function, which equals one if $i = k$, and equals zero otherwise; $z_j(t)$ represents the jth input to neuron i. The need for this term can be explained as follows: At time t, neuron i receives the value of input $z_j(t)$ multiplied by weight w_{ij}. At $t+1$, neuron k receives the output of neuron i as an input. Note that in this case there is no intermediate neuron l for weight w_{ij} to influence neuron k through, hence no p_{ij}^l term. If neuron i and neuron k are the same, but at different time steps, weight change Δw_{ij} affects neuron k's output indirectly through changing its output directly during the previous time step.

The change in the output of neuron k in respect to a change in the weight w_{ij} can be represented $p_{ij}^k(t+1)$, by the equation

$$p_{ij}^k(t+1) = \frac{\delta y_k(t+1)}{\delta w_{ij}} \tag{17}$$

Thus, equation (17) can be rewritten

$$p_{ij}^k(t+1) = f_k'(t)\left[\sum_{l \in U} w_{kl}\, p_{ij}^l + \delta_{ik}\, z_j(t)\right] \tag{18}$$

The p^k_{ij} term is implemented in the C code as a n x n x $(m+n)$ matrix (p matrix), and is used to update the n x $(m+n)$ weight matrix. **This is the direct cause of the $O(wn^2)$ complexity of this algorithm**, and the reason why RTRL too slow to train for other than small problems, limited in the number of outputs or of hidden nodes. Each weight is updated based on its effect on all of the neurons in the net. To avoid this, the net can be subgrouped in such a way that when weight w_{ij} is updated, it is only based on its effect on the neurons within its group.

In the subgrouped RTRL implementation used for this thesis, the number of groups in the net is equal to the number of network outputs. Because of this, the p matrix (p^k_{ij}) becomes an s x n x $(m+n)$ matrix, where s equals the number of nodes in each of the subgroups in the net. The size of the p matrix has been reduced by a factor of g, which is the number of groups in the net.

In the subgrouped RTRL algorithm, equation (18) becomes

$$p_{ij}^k(t+1) = f'(t)\left[\sum_{l \in U_g} w_{kl}\, p_{ij}^l + \delta_{ik}\, z_j(t)\right] \tag{19}$$

where $k \in U_g$, $l \in U_g$, $i \in U_g$, and $j \in I \cup U$. U_g is that subset of the recurrent neuronal outputs that belongs the group containing neuron i. In other words, the $\sum_{l \in U_g} w_{kl}\, p_{ij}^l$ term represents the summation of the recurrent (neuron l) inputs to neuron k, where neuron l is from within neuron k's subgroup, times the change in neuron l's output caused by changes

to weight w_{ij} during a previous iteration. The effect that weight w_{ij} has on neurons and weights outside the subgroup *are not calculated.* The consequence of this change is that the p matrix is smaller, the net runs approximately g^2 times faster. Zipser(20) hypothesized that the subgrouping of the network may impact net accuracy, but believed this can be compensated for with the addition of extra hidden nodes. He also stated that the time delay caused by the additional nodes should be more than compensated for by the speedup caused by the subgrouping.

All of this theory being said, the Update subroutine begins by first calculating the delta weights (Δw_{ij}) based on the p matrix calculated during the last iteration. In the RTRL algorithm, this weight update is derived from equation 16 as

$$\Delta w_{ij} = \alpha \sum_k e(k) p_{ij}^k \tag{20}$$

where $i \in U$, $j \in U \cup I$, $k \in U$, and α is the learning constant. In the subgrouped RTRL algorithm, this becomes

$$\Delta w_{ij} = \alpha e(k) p_{ij}^k \tag{21}$$

where $i \in U_g$, $j \in U \cup I$, and $k \in U_g$. Thus only the error at each group's output node drives the weight updates for the weights associated with that group.

Next, the subroutine calculates a new p matrix based on the above algorithm, and saves the new p matrix for the next weight update.

3.3 Network Parameters

The RTRL algorithm, as implemented by Capt. Randall L. Lindsey(7), was able to perform several time dependent tasks quite well. These tasks, however, required only one or two outputs. When research on this thesis was begun, it was quickly determined that the algorithm as outlined in Lindsey's thesis was not appropriate for some of the larger, more complex tasks. Processing time required for training on phoneme broad classes for more than one voice was measured in days. The outputs of the network would tend to lock onto zero or one, even if the output was in error. To avoid this problem, research

into means of improving the training time and accuracy of the network was initiated. This led to the exploration of the subgrouped network as proposed by Zipser(20), as well as several other methods of manipulating network performance.

The following is a discussion of the network parameters or algorithms added to modify the learning behavior of the implementation of the subgrouped RTRL network used for this thesis, in an attempt to improve network learning speed and accuracy. Evaluation of the initial learning rate, momentum, minimum sigmoidal derivative, teacher forced learning, and weight update skipping error threshold parameters were performed using a Payton algorithm (8) processed TIMIT voice file. The voice file selected has 389 data points of 20 inputs and 6 outputs, each output corresponding to one of six broad classes of phonemes. Each training run using these parameters was performed on ten networks with different initial random weights, and the results of the training runs were averaged for a composite graph of the network output accuracy. The graphs showing the composite accuracy for the above parameters are shown in Chapter IV.

One network parameter was evaluated without using the voice file data. It allows the net to treat the training data as continuous between the end of one epoch and the start of the next, and is discussed in section 3.3.6. This capability was added to address a byproduct of the way in which the RTRL algorithm learns, and so is discussed using the type training problem in which this byproduct can be observed.

3.3.1 Variable Learning Rate

The subgrouped RTRL network used for this thesis reduces the learning rate (alpha) by a factor of ten whenever the network error rises more than 1% over the minimum error reported to that point, or if the difference in error between the current epoch and the previous epoch is less than 0.0000001. This is done to prevent the network from becoming unstable if the learning rate is too high, and to improve the network error minimization when the net error has reached a plateau.

Setting the learning rate at a high or low level at the beginning of training has a definite impact on the network's ability to learn a task over the training period. Set the rate too high, and it immediately adapts to the inputs at time *t*, forgetting previously learned behavior and therefore losing it's ability to generalize. Start with too low an initial learning rate, and the net learns slowly and may become stuck in a local minimum.

To observe the effect of the initial learning rate on network training, the net was trained using the Payton processed voice file for 200 epochs, with initial learning rates of 0.1, 0.01 and 0.001. The network configuration was the 20 neural activity level inputs produced by the Payton algorithm, 6 sigmoidal outputs, and 12 sigmoidal hidden nodes. The training output was delayed two time steps. For this problem, the best learning results were obtained using an initial rate of 0.01. The differences in network performance caused by different initial learning rates are discussed in Chapter IV.

3.3.2 Momentum

The use of momentum to speed up the learning of a backpropagation network is well established (14). Use of momentum tends to dampen out the oscillations in network accuracy during learning, and carry the net down the averaged out path to an error minimum. To add momentum to the network, the delta weights are simply multiplied by the momentum factor after the weights are updated. This is summed with the next calculated set of delta weights, to allow the carry over a portion of the weight update from time *t*-1. The momentum factor is a parameter read by the network during initiation.

The impact of using momentum was measured by training the subgrouped RTRL net using momentum set at 0, 0.5 and 0.9, with a network configuration of 20 inputs, six sigmoidal outputs and 12 sigmoidal hidden nodes. The network with a momentum of 0.9 demonstrated the highest accuracy and lowest error during training, followed by the net with 0.5 momentum factor. This indicates that momentum does improve training performance for this problem. Further discussion of this test is provided in Chapter IV.

3.3.3 Minimum value for output derivative factor

For an output y(*i*), the derivative of the sigmoid transfer function is y[*i*](1-y[*i*]). One common problem encountered when using neural networks for categorization of inputs is that the derivative of the sigmoid output function tends towards zero when the output approaches zero or one.

$$(y[i])(1-y[i]) = (1)(1-1) = (0)(1-0) = 0 \qquad (22)$$

Even if the output is wrong, the error feedback used to update the weights is zero or very small. This can cause an output to "hang" or latch on a wrong value, slowing down learning tremendously. Van Ooyen and Nienhuis (16) proposed the use of a different energy equation,

$$E = -\sum\sum\left[t_j \ln z_j + (1-t_j)\ln(1-z_j)\right] \qquad (23)$$

where t_j represent the desired nodal output, and z_j represents the actual output of node j. When this function is used, the partial derivative of the error function contains the inverse term to the sigmoid function derivative, canceling it out. Thus during weight updates the error at the output is fed back directly without the sigmoid derivative term, avoiding the latching of the neuron in the wrong state during training. Rather than redefining the error function for the subgrouped RTRL network however, a similar effect was gained by setting a minimum value for the sigmoid derivative of the output neurons. When the derivative falls below the minimum set value, the set value is used for the updating the p matrix (equation 24). Above the set value, the sigmoid derivative value is used.

$$p_{ij}^{k}(t+1) = f'_{\min}\left[\sum_{l \in U_s} w_{kl}\, p_{ij}^{l} + \delta_{ik}\, z_j(t)\right] \qquad (24)$$

This one change appears to have caused the biggest improvement is learning effectiveness, compared to the other variables used to manipulate the network.

3.3.4 Teacher forced learning

Williams and Zipser(17) stated that learning could be accelerated if teacher forced learning is used. Teacher forced learning involves replacing network output values from time t with the desired values (after computing the error), which are then used as the recurrent inputs at time $t + 1$. This helps to train the network faster for some problems, as the net does not have to unlearn weights as the recurrent output values transition from incorrect to correct values during training. In some cases however, this approach backfires. When the net is being tested with new data, any erroneous outputs are fed back as inputs. As the network weights were trained to work with the "correct" outputs, erroneous outputs can make the net unstable. In this case, teacher forced learning causes the correct response to represent an energy repellor rather than attractor. The RTRL code was modified to allow for teacher forced learning if the user selects it. This is set using a flag in the parameters file access by the net upon initialization.

3.3.5 Skipping weight updates for learned outputs

Allred and Kelly(1) proposed performing backpropagation of the error during training only when the error for a neuron was greater that the learning rate value squared (α^2). As the network error decreases and the number of data iterations that skip error backpropagation passes 90%, α is decreased. This idea was incorporated into the recurrent network code by feeding a parameter to the network during initialization which sets an error threshold for weight updates. When the output error is below the threshold, the weights are not updated. Since the calculation of the weight updates in the RTRL algorithm is the most time consuming part of training the network, skipping weight updates holds the potential for speeding up network training considerably.

3.3.6 Continuity of Recurrence Between Epochs

During each iteration of the RTRL training the net output and hidden node values are forwarded as inputs for the calculation of the next iteration. The exception to this is at the end of the epoch, when the output value of the neurons (and the p matrix) are replaced with zeroes. When training the RTRL network for some functions, it was found to be better not to zero out the net outputs at the end of each epoch. This is due to the discontinuity the zeroing of the outputs induces at the beginning of each epoch. An example of this phenomenon can be seen when training the network to emulate a low pass Butterworth filter (paragraph 3.4.3). At the initial iteration of the epoch (t=0), if the training data is zero and the output from the previous epoch has been zeroed, the net sees only the bias as a non zero input (Figure 10).

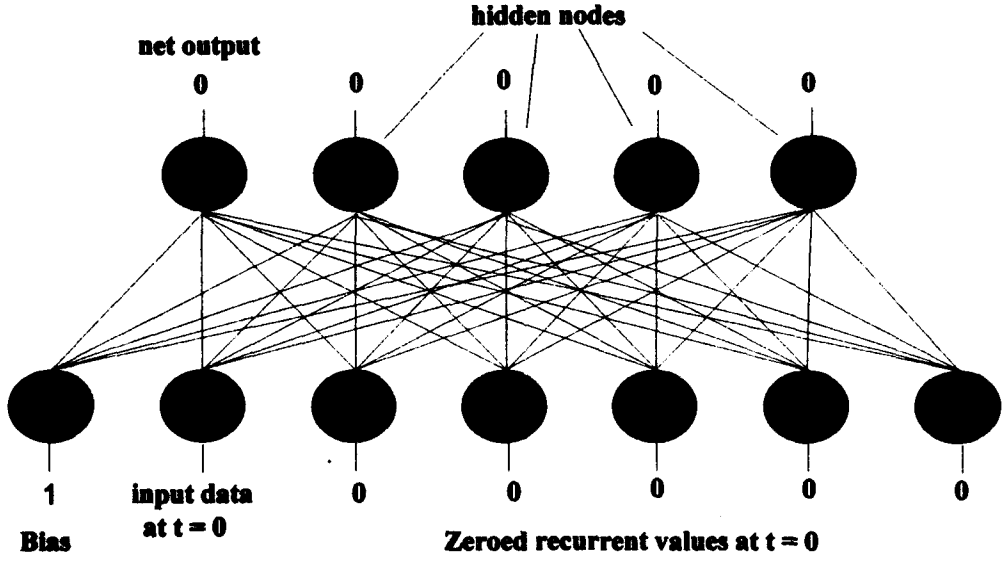


Figure 10: The RTRL network (1 output, 4 hidden nodes) at t=0, after zeroing the output values from the last iteration of the prior epoch

The weighted bias drives the output values of the RTRL neurons, which are fed back into the net during the next iteration. The net treats this input as an impulse, and generates the filter's impulse response (Figure 10).

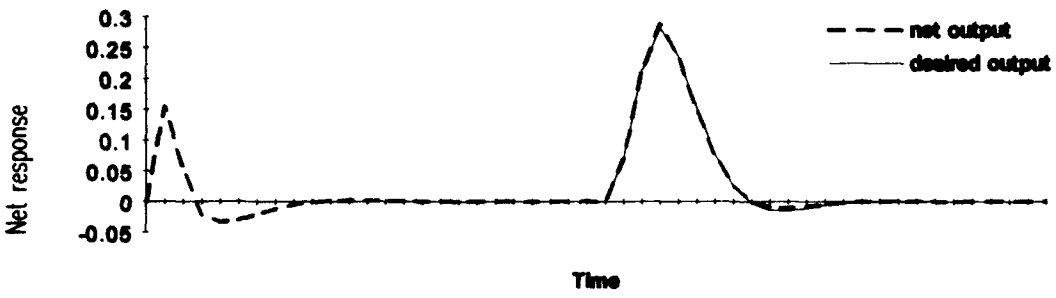


Figure 11: The recurrent network shows the Butterworth filter's impulse response at the beginning of the epoch, after training with inputs zeroed at the beginning of each epoch

The enabling of this continuity option causes the output from the final iteration of the previous epoch to be forwarded, as the RTRL net does in all other iterations, as inputs into the calculation of the next iteration, the first of the current epoch. One complication to this option is the fact that all RTRL training files have some delay imposed in the network outputs, due to the time dependent nature of the network. To make the training on the data truly continuous, the desired outputs generated by the last data iterations in the training file must be placed as training outputs at the beginning of the file. If the outputs are delayed for two iterations for example, the outputs associated with the last two data iterations must be placed as the desired output with the first two iterations of data at the start of the file.

3.4 Subgrouped RTRL Functional Capabilities

To demonstrate the functional equivalence and/or improvements gained using the subgrouped RTRL code over the original RTRL program explored by Capt. Lindsey, several of the same tests were performed as were described in his thesis(7). The repeated tests were the Exclusive OR problem, the internal state problem, and the Infinite Impulse Response (IIR) filter simulation. The subgrouped RTRL was also tested by training it to categorize the phoneme groups in a sample of digitized voice that had been pre-processed by the Payton(8) algorithm. During the training/testing of the network on the pre-

processed voice data, the differences in performance in training speed and accuracy for this task between the subgrouped RTRL and the original RTRL code were measured

.

3.4.1 Exclusive OR (XOR)

The Exclusive OR problem is a classic test of the performance of a neural network, as it requires the identification of two distinct and separate areas in the solution space. This is a task beyond the capabilities of a single layer network. From appearances, the RTRL network seems to be a single layer network, and therefore incapable of learning an XOR solution.

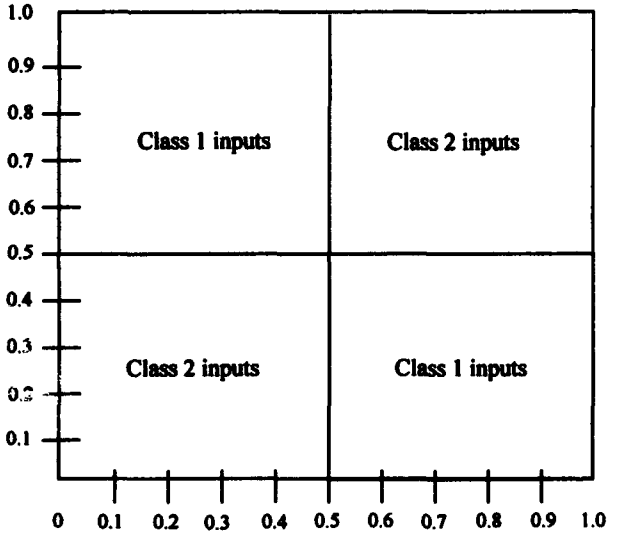


Figure 12: For the XOR function, valid input values can not be isolated into one contiguous area.

The hidden nodes of an RTRL network, unlike a standard backpropagation network, do not feed directly into a higher layer during the processing of a data set at the input layer. Instead, they feed into the output layer, and to themselves, during the next iteration. This temporal means of connecting the hidden nodes to the output nodes enables the network to solve the XOR problem. To allow for the temporal delay in passing the hidden node outputs to the output nodes however, the desired output of the network must be shifted ahead in time.

The network configuration used to solve the XOR problem with the subgrouped RTRL network was identical to the network used by Lindsey's code, i.e. two external inputs, one sigmoidal output and four hidden sigmoidal units. The ones and zeros used as inputs were generated randomly, and the training output for the XORed function of the

two inputs delayed by two time steps. Using 1024 training vectors the network was trained over 20 epochs, and then tested using the trained network weights on a test XOR sequence.

The training of the networks was repeated using non-integer training values between 0 and 1, with the range 0 to 0.5 treated as a zero input, and 0.5 to 1 equivalent to a input of 1 for the determination of the XOR output. Using a two input, one sigmoidal output node and 5 hidden node configuration, the net was trained for 300 epochs through the 512 training vectors. The trained net was then tested on a 1024 vector non-integer test set.

While the net scored perfectly on the integer portion of this test, it only scored in the 91+ percentile when trained and tested on non-integers. Interestingly enough, the misses were not at the boundary data values where one would expect. The complete results of these tests are discussed in Chapter IV.

3.4.2 Internal State

Backpropagation networks have no temporal memory; they only train or respond to the data at the network inputs during each iteration. This property makes these networks unsuitable for training on patterns that occur over a series of iterations. To recognize a pattern over time, a network must maintain some form of internal memory or state over one or more time intervals. A test of this function, as discussed by Williams and Zipser(17) and documented in Lindsey's thesis(7), consists of presenting the network with data vectors of 4 inputs labeled *a, b, c* and *d*. Within each data vector one input randomly selected is valued at 1, the others are zero. The output of the network is normally zero, except for the interval immediately after a valid *b* input (*b*=1) follows a valid *a* input. When this occurs, the desired network output is 1 for one time interval. Inputs *c* and *d* have no effect on the desired network output. Training and test files for

this problem were created using different random number seeds, so that the order of inputs, internal states and intervals between valid a and b inputs were varied.

The network configuration for the internal state test was four inputs, one sigmoidal output, and one sigmoidal hidden node. When tested, the net apparently had learned this task perfectly, as had the original RTRL algorithm. The discussion of the results of the internal state test is presented in Chapter IV.

3.4.3 Second Order IIR Lowpass Filter Simulation

In this test, the subgrouped RTRL network was trained to simulate a second order low bandpass Butterworth filter. The filter algorithm used to produce the training and test output data for the network is described by the equation

$$y[t]=0.0676(x[t]+2x[t-1]+x[t-2])+1.1422y[t-1]-0.4124y[t-2] \qquad (25)$$

The inputs to the network, and to the above algorithm, consisted of a several different data series: a set of random values between -1 and 1, a set of impulses (1 followed by a string of zeros), a step function (0 0 0 0 0 1 1 1 1 1 1 1 1) and a sampled cosine wave. The network was trained on the filtered series of random values, followed by training on a filtered series of impulses. The trained network was then tested on the filtered impulse, step function, cosine wave and random number data sets. This training approach differs from the one described in Lindsey's thesis(7), where the filtered impulse series was used for training. The method used for the subgrouped RTRL resulted in faster training and higher accuracy after training. The net configuration consisted of one input, one linear output node, and one sigmoidal hidden node.

The net learned to emulate the Butterworth filter with good fidelity, with only minor deviations from the desired response. The details and accuracy of the network's filter emulation is discussed in Chapter IV.

3.4.4 RTRL Versus Subgrouped RTRL Performance

The subgrouped RTRL network was evaluated for performance by comparing how quickly both of the RTRL algorithms (original and subgrouped) performed 10 training epochs using 0, 6, 12, 18, 24 and 30 hidden nodes. Training was performed on a Sun Sparc 10 workstation, and processing time was obtained using the UNIX *time* command, which reports how much CPU time was dedicated to the process in question. This allows time data to be taken without concern over varying CPU workloads.

The training file consisted of 389 data vectors (20 inputs and six outputs) from a single voice data file. The input data used to train the networks consisted of digitized voices derived from the TIMIT voice database, which have been processed through the Payton(8) auditory model algorithms. This training data required the RTRL networks to differentiate between six classes of phonemes (nasals, vowels, stops, fricatives, silence and liquid-glides). Training runs with the subgrouped RTRL network were performed twice, first without allowing weight update skipping, and the second time with the error threshold for performing weight updates set at 0.00001.

The subgrouped RTRL network trained in substantially less time than the RTRL algorithm, but the RTRL network showed a higher average accuracy in identifying the phoneme classes as compared to either subgrouped RTRL network. Skipping weight updates in the subgrouped RTRL network incurred a small penalty in network error, but depending on the application, this may be offset by the increase in training speed. The time required for training, and the increase in processing speed for these network configurations, is discussed in Chapter IV.

3.5 Applications

Since the strength of the RTRL algorithm is in the ability to deal with data that changes over time, the subgrouped RTRL was applied to two time dependent problems. The first deals with testing the predictive ability of the network, using the opening value

of the pound at the London Exchange for training and then testing the network. The other problem deals with classification of time dependent data; image classification based on feature changes over time.

3.5.1 London Exchange Prediction

It is the dream of every financial analyst to possess a sure method for predicting the value of a stock, commodity or currency at some point in the future. One potential method for this, evaluated in this thesis, is to present a time dependent neural network with a sequence of values (daily pound exchange rates) over time, with desired output being the value at some point in the future.

The data used to train the network was derived from the London Exchange, and consisted of the opening exchange value of the pound over a period of one year. The desired output provided to the network was the same data sequence, shifted in time one day. At any particular time t, the desired output of the net would be the next input value at time t+1. The network consisted of one input, one linear output, and three sigmoidal hidden nodes. It was trained for 500 epochs, with an initial learning rate of 0.0001. The net was then tested on the opening exchange rates for a different year.

While the net learned to closely match the desired response, examination of the plotted net output shows that it consistently lagged behind the desired (future) output. This plot of the results, and the future of this network as a financial analyst, is discussed in Chapter IV.

3.5.2 Vehicle Image Classification

For the application of the subgrouped RTRL network to the problem of image classification, the net had to associate sequences of single value codewords with the vehicle the sequence had been derived from.

The sequences of codewords or feature vectors used to train and test the network were derived from the 3 dimensional CAD representations of five different vehicles: an M-60 tank, an M35 truck, a BTR60 armored personnel carrier, a T62 tank, and an M2 infantry fighting vehicle. The CAD images of each vehicle were captured from multiple points above and around the vehicle representation, to uniformly cover possible perspective points for viewing the vehicle. The multiple images of the five vehicles were processed(3), and the features extracted into 64 possible states, represented with codeword values of 0 - 63. Sequences of the codewords represented a series of discrete perspectives or image frames of a vehicle, changing over time as the viewer perspective point changes.

The 64 codewords did not in themselves represent any of the vehicles; each may be found in a sequence associated with any of the five vehicles. Instead, it is the sequencing of the codewords that differentiates between the vehicles.

The data files associated with each of the vehicles contained 200 sequences of codewords of four different lengths; 50 sequences each of 14, 16, 18 and 20 codewords. The five data files were combined, with each sequence associated with a vehicle category. Categories were represented by six network outputs, one for each of the vehicles plus one for the "header" information between the sequences. The codewords were represented to the network in binary form, with the header assigned a value of one, and codewords 0 - 63 presented as 0 0 0 0 0 1 0 to 1 0 0 0 0 0 1. The order of the codeword sequences in the datafile was randomized, and the first 90% of the sequences were used as training data for the network.

The network consisted of seven inputs (binary representation of codewords), six sigmoidal outputs, and six sigmoidal hidden nodes. The desired output values used to train the network were delayed two time periods, so that the network "saw" the desired output at time t that corresponded with the input presented at time t - 2. The initial learning rate of the network was 0.01, momentum was set at 0.98, and the net was trained

for 1000 epochs. After training, the network was tested on the remaining 10% of the randomized datafile. The net scored a 99+% accuracy in identifying sequences with the correct vehicular image. Chapter IV expands on the results of this application, with a discussion of the network's performance.

3.6 Summary

The subgrouped RTRL algorithm, and the modifications made to the algorithm in the development of the C code used for this thesis, were described. The methodology for the testing of the subgrouped RTRL was also discussed. The results of these tests demonstrate how the performance of the subgrouped RTRL algorithm relates to the RTRL algorithm described by Lindsey(7), as well as how the network performs at prediction and classification based on time varying phenomena. Chapter IV contains the results and discussion of these tests.

IV. Results and Discussion

The history, theory and testing of the subgrouped RTRL algorithm were discussed in Chapter III. This chapter reviews the operating parameters of the network and their effects, and the results of the tests conducted to demonstrate the network's abilities.

The subgrouped RTRL was tested to quantify the impact of the various operating parameters (momentum, minimum derivative factor, teacher forced learning, weight update skipping, continuity between epochs) that had been added to the net to enhance performance. The net was then tested to determine how the subgrouping of the network caused the capabilities of the network to change from that of the RTRL algorithm, using the performance described in Lindsey's (9) thesis as a reference.

The problems presented to the subgrouped RTRL net as potential applications were twofold: testing the net as a predictor using the daily opening values of the British pound as training data; and testing the net as an image classifier based on learning vector quantized codewords derived from vehicle images.

4.1 Network Parameters

To demonstrate the effects of the different operating parameters on the performance of the subgrouped RTRL network, several of the factors (initial learning rate, momentum, minimum derivative factor, weight update skipping) were varied during network training. The training file used was a Payton (8) model processed digitized voice file, derived from the TIMIT database. This file contained 389 data vectors, and was set up to train the net to provide six outputs, one for each of the broad phoneme classes.

This data set was chosen as an example because it was difficult enough that the network does not completely solve it, reaching a maximum accuracy of approximately 80%. It was believed that this environment would help to demonstrate the effects of the network's parameters, more so than a problem where the error rapidly drops to a low

value. The initial learning rate (alpha) for the momentum, minimum derivative factor, and weight update skipping trials was set at 0.01, with a network configuration of 20 inputs, 6 sigmoidal outputs, and 12 sigmoidal hidden nodes. Training time was set at 200 epochs. The default settings of the parameters (aside from those varied for the test) are:

Initial learning rate (alpha) = 0.01

Momentum = 0.0

Sigmoidal derivative minimum = 0.01

Output is sigmoidal

No teacher forced learning

Weight updates skipped if error <= 0.0

End of training epoch not continuous with beginning of next epoch

The effect of making the training data and network operation continuous over different epochs (continuity between epochs) is demonstrated while training the net to emulate the impulse response of a Butterworth filter. This was due to the fact that this option was added to eliminate a phenomenon found while training the network for this task.

Each line on the graphs shown in this section average the results of ten training runs, using different initial values of the randomized weights. Reported net accuracy was based on matching the desired output category with the network output with the highest activation value.

4.1.1 Initial Learning Rate

The value of an adjustable learning rate can be seen using the subgrouped RTRL code evaluated in this thesis. In many cases after the network error levels off, a cut in the learning rate produces an immediate improvement in net accuracy and error. While the

net will lower the learning rate if the net error increases, a high initial learning rate is not necessarily harmless to the overall learning behavior of the net during training. If the rate is initially too high, it may push the weights to a state that the net must recover from after the rate is decreased. If the rate remains too high the net error tends to climb over time, in some cases to the point of creating overflow errors.

The effect of the initial learning rate on net performance was examined by training the subgrouped RTRL net with three different alphas at initialization (0.1, 0.01 and 0.001). The configuration of the network was 20 inputs, 6 sigmoidal outputs and 12 sigmoidal hidden nodes. Figure 13 shows how the different initial learning rates impacted the network accuracy during training. As can be seen from this graph, the best performance was achieved with an initial learning rate (alpha) of 0.01.

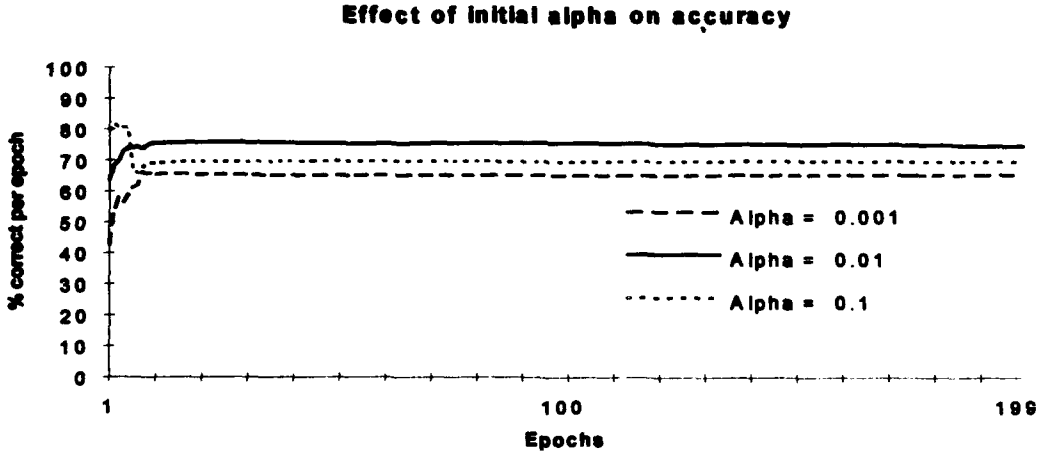


Figure 13: The impact of different initial learning rates on network accuracy during training. The average standard deviation of the data was 6.13.

The higher accuracy reported at the start for the initial learning rate of 0.1 was caused by the net rapidly changing its weights to adapt to the most recent inputs. This causes the net to be correct at time *t*, but after passing time *t* the weights would change enough that the same inputs might produce different and erroneous outputs. When the net training is halted and tested while in this reported higher accuracy state (circa 5

epochs) the net performs poorly, and the test reports a low accuracy result. This is due to the fact that the test uses fixed weights, rather than the rapidly adapting weights generated by training with a high alpha that creates temporary error minima as it goes.

4.1.2 Momentum

The inclusion of a momentum term (μ) as a means of increasing the learning rate of a neural net is a well understood mechanism for improving learning performance. By retaining a fraction of the weight update from the previous learning iteration and adding it to the current weight update, weight changes tend to continue along the same direction over time. This has the tendency of damping oscillations in the network as it learns, and maintaining the progression of the net to an energy minimum. The effect of the momentum term in the broad class phoneme identification problem is shown in Figure 14.

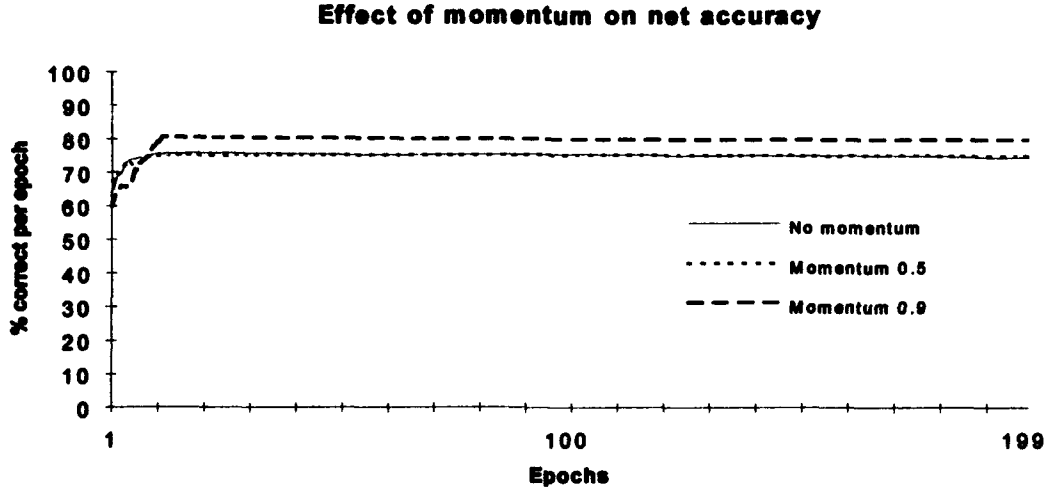


Figure 14: The accuracy of the network over 200 epochs is shown, with the momentum term set at 0, 0.5 and 0.9. The average standard deviation of the data was 3.79.

The network exhibited a higher accuracy during training with a momentum of 0.9. Thus the apparent benefit of momentum appears to work with RTRL type networks as well as for standard backprop networks, at least for this type of problem.

4.1.3 Minimum value for output derivative factor

The establishing of minimum value for the sigmoidal derivative factor in the weight update formula has a profound effect on the learning rate for a certain class of problems. This class includes those problems for which the desired output(s) of the network are either zero or one, usually to signify Boolean decisions (yes or no) and in determining membership in categories. Figure 15 shows how setting the minimum level for the sigmoidal derivative affected the learning rate of the subgrouped RTRL network when solving the broad phoneme category problem.

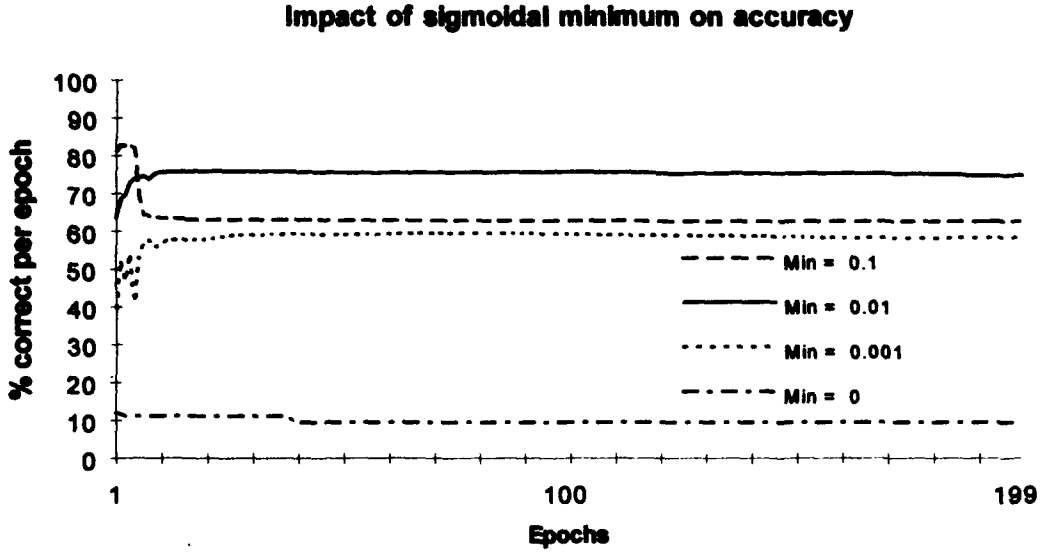


Figure 15: The effects of setting a minimum sigmoidal derivative factor for error backpropagation. *Note how the network did not progress when a minimum factor was not set.* The average standard deviation for the lines was 5.25.

The addition of the minimum derivative term to the RTRL network was perhaps *the* most effective modification in terms of enhancing learning for any categorization problem. Prior to this modification, the RTRL network would "latch" and not progress unless the learning constant was set very low. This reduced the effective learning rate to an unacceptable level, and the network appeared to be unsuitable for differentiating between several categories.

It is also noteworthy that the runs with the highest sigmoidal minimum set (minimum=0.1) reported a higher accuracy at first, which then dropped in much the same way as the nets training with a high initial learning rate. Again, when tested after a few (~5) training epochs these nets report a low accuracy, because the net was adapting too quickly to the inputs. Based on having applied this minimum factor to a wide range of different problems, the optimum level for the derivative minimum appears to be on the order of 0.01 for almost all training problems where a sigmoidal network output is required.

4.1.4 Teacher forced learning

As stated in Chapter III, teacher forced learning can cause the network to train faster, but may reduce the network accuracy once the constraint of passing only the correct outputs back as network inputs is removed, such as during testing of the trained network. To demonstrate the use of teacher forced learning therefore, not only must the learning rates with and without teacher forced learning be examined, but the accuracies of the network after training must be checked as well. The differences in reported accuracy during training is shown in Figure 16.

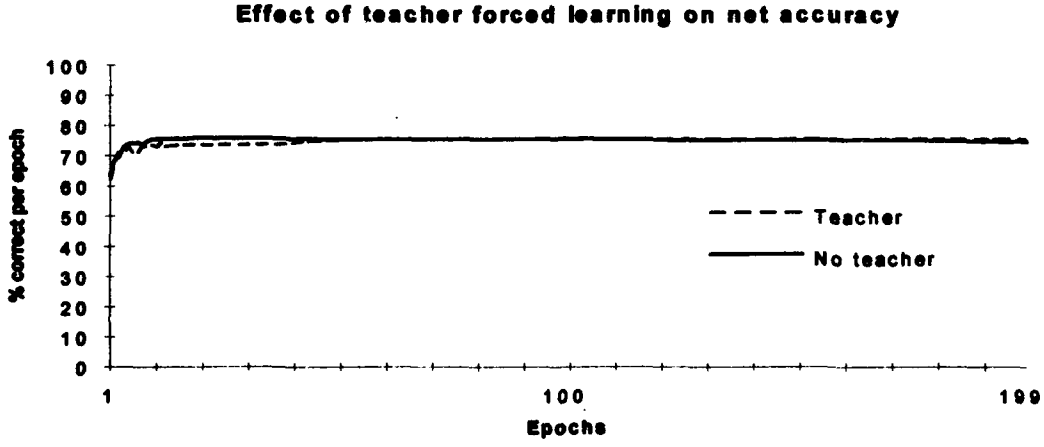


Figure 16: A comparison of learning rates with and without teacher forced learning. The average standard deviation for the data used to plot the graph lines was 4.98.

As can be seen from the Figure 16, the addition of teacher forced learning for this problem had little impact on the learning rate of the network. Testing the network on the training data file showed a 70.4 percent accuracy (σ = 17.69) in identifying the phoneme groups with teacher forced learning, and a 79.4 percent accuracy without. (This was in part due to an outlying test result of 20.3% for one of the ten networks trained using teacher forced learning, pulling the average down. Without this outlying value, the teacher forced learning nets tested at an average 75.85% accuracy.) For this type problem teacher forced learning provided no real gains, but instead induced a loss in phoneme group recognition performance.

4.1.5 Skipping Weight Updates forLearned Outputs

The computation of the p matrix is the most time consuming routine in the RTRL network, and therefore the primary driver for the investigation of optimization methods to speed up learning for this algorithm. The addition of the weight (and p matrix) update skipping can cut the time required for processing each epoch of data up to 50%, significantly improving the training rate of the network.

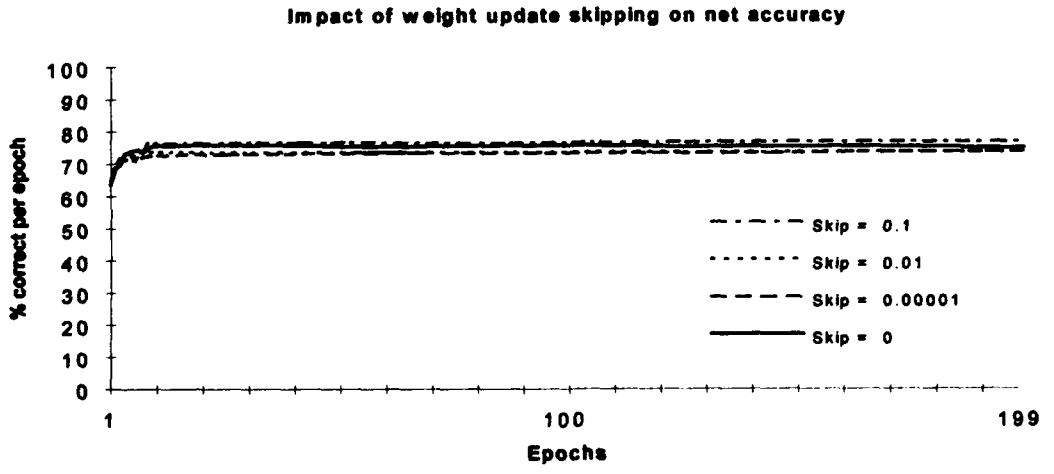


Figure 17: The effects of skipping iterations during learning when error is below the skip threshold. The average standard deviation for the lines was 4.29.

As can be seen in Figure 17 however, skipping weight updates for outputs with low errors does impact the accuracy of the network to some small extent. Paradoxically, the accuracy shown by the nets training with a 0.1 error threshold shows a higher overall accuracy during training than training with lower error thresholds. Skipping more of the weight updates may allow the net to focus more on inputs that are outside the average location in the input space, or perhaps causing the net to learn to classify some inputs that may be in the minority, and therefore normally not caught by the net. The effect of changing the error threshold on network accuracy was the primary reason why the skip threshold was made to be changeable by the network user; the user can determine where the error threshold should be set.

4.1.6 Continuity of Recurrence Between Epochs

In paragraph 3.3.9, the problem caused by zeroing out the network outputs at the end of each training epoch was discussed. The effect of not zeroing out the net outputs was evaluated by training the RTRL network to emulate a low pass Butterworth digital filter. The net was trained using the methods detailed in paragraph 3.4.3, with and without the data continuous at the ends of the training epochs. Each net was trained first on a sequence of random floating point values (between -1 to 1) with their low pass filter response, followed by training on impulses (0 0 0 1 0 0 . . .) coupled with the filter's impulse response. The nets were then tested on the impulse response training data. The reactions of the networks to the impulse are shown in Figure 18.

As can be seen in Figure 18b, removing the discontinuity between the epochs removed the additional impulse response, shown in Figure 18a. Using this option caused the network to train to a closer match of its output to the desired response, to the extent that the lines (desired vs. output) are almost indistinguishable.

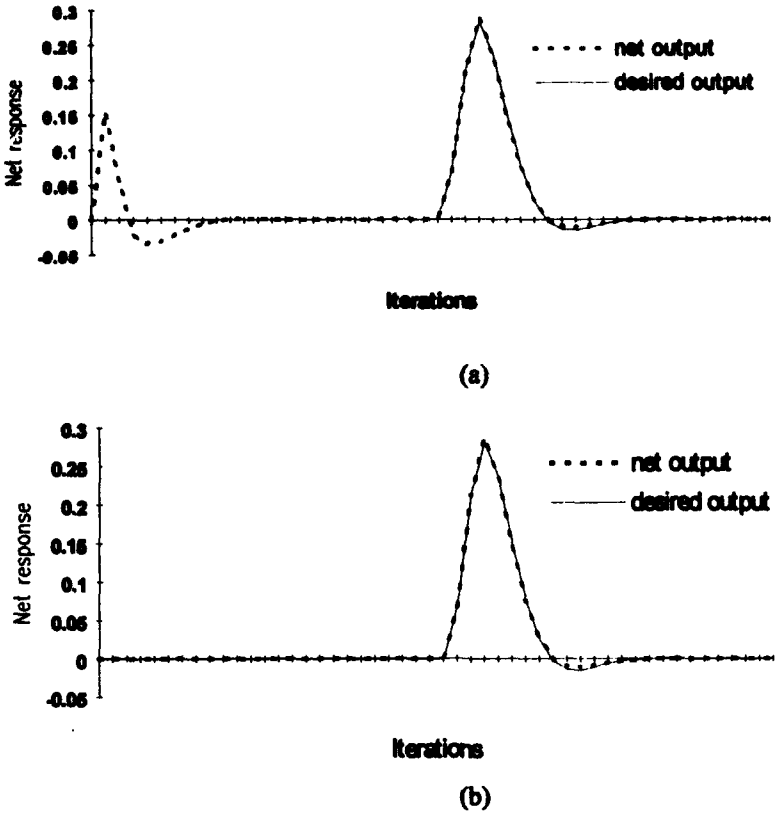


Figure 18: These charts show the impulse response of the RTRL network without (a) and with (b) continuity between epochs. The net was trained to emulate a Butterworth filter.

4.2 *Subgrouped RTRL Functional Capabilities*

So far in this chapter only the parameters of the network have been discussed. These parameters can more or less enhance the learning efficiency of the RTRL network program, but do not necessarily demonstrate the subgrouped RTRL network's characteristics or capabilities. Subgrouping the RTRL network could have negatively impacted the ability of the network to perform various functions. This section of the thesis therefore evaluates the subgrouped RTRL network's properties and abilities, as compared to the RTRL algorithm described in Lindsey's(7) thesis. Several tests described in that thesis were therefore used as a benchmark to measure the impact of subgrouping the network.

4.2.1 Exclusive OR

As in Lindsey's(7) thesis, the first problem to be examined to demonstrate the capabilities of the network is the eXclusive OR (XOR). As described in section 3.4.1, the net was trained using 1024 binary training vectors, with the two inputs, one sigmoidal output neuron and four hidden neurons. The outputs provided in the training file were delayed by two time steps. After 20 epochs, the network established 100% accuracy with a mean squared error of 0.030. The criteria for a valid response from the network was an error of less than 0.5, meaning that the mean squared error had to be less than 0.125. The network was then tested on a separate binary XOR data set created with a different random number seed, and was found to have a 100% accuracy on the test file as well. This demonstrated that for binary (0 and 1) data, the net was able to generalize the XOR problem.

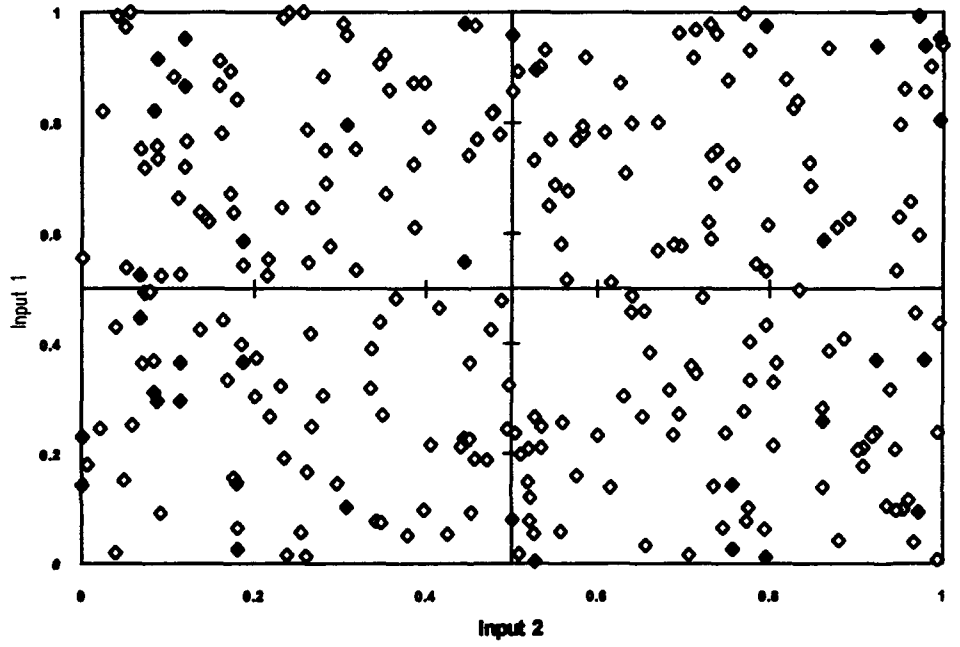


Figure 19: Plot of the subgrouped RTRL network's hits and misses for the third analog XOR test set. Network accuracy for this test set was 91.2%. Hits are designated with open diamonds, while misses are show with filled diamonds.

The next step in training the network to recognize the XOR problem was to use an analog test set, with two input data values between 0 and 1. If one input was greater than

0.5 and the other less than 0.5, the output (delayed two time steps) was 1, otherwise the output was 0. After training the network over 300 training epochs using 512 training vectors, the net achieved an accuracy of 99.6%. The net was then tested on three analog XOR test files, and received accuracies of 92.6%, 94.5% and 91.2%. This corresponded with the results seen by Lindsey(7).

As can be seen in Figure 19, the misses in the third test file (91.2% accuracy) do not correspond to the axis between the decision areas, but are scattered throughout the test space. This implies that the net is solving a temporal path through the test data, rather that differentiating each pair of inputs as valid or not.

It was interesting to note that neither the original RTRL code nor the subgrouped RTRL code could solve the XOR problem for an output time delay of less that 2 time steps. This may indicate that to solve the XOR problem, the data must recursively pass through the hidden nodes at least twice, effectively solving the problem with two or more hidden layers. Because of this, it may be feasible that for any problem, a balance must be struck between delaying the outputs long enough to use multiple hidden layers in the problem, and having the outputs close enough in time to the associated inputs that the net can infer a causal connection between the two.

4.2.2 Internal State

The ability of the subgrouped RTRL network to internalize a time dependent state was demonstrated by the test described in section 3.4.2 of this thesis. The net was trained using four binary inputs (**a, b, c** and **d**), with the desired response of recognizing the occurrence of the first valid **b** input (value = 1) after a valid **a** input. After training on the 95 input vectors, the subgrouped RTRL network obtained an accuracy of 100%, given a decision threshold of 0.5 for valid (high) versus nonvalid (low) network outputs. Figure 20 shows the network output over the 95 training vectors versus the desired output.

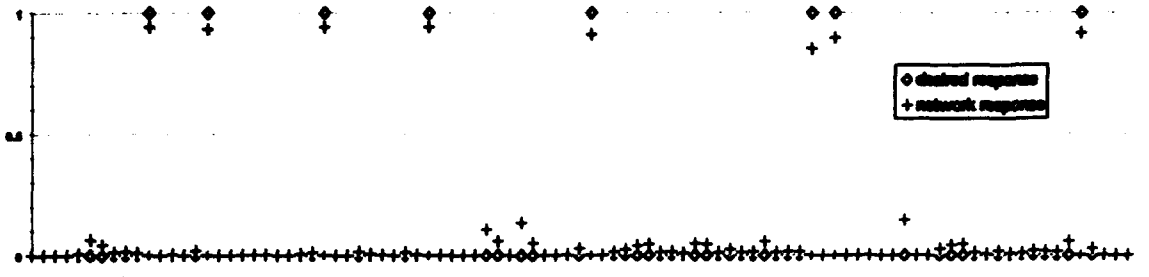


Figure 20: Internal State Training Results

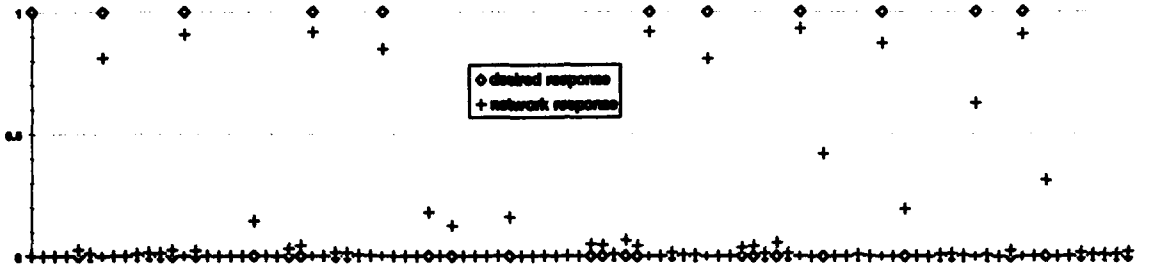


Figure 21: Internal state Testing Results

The network was then tested on a different internal state data file, and the subgrouped RTRL net again demonstrated a 100% accuracy level (Figure 21). The network therefore was able to generalize the solution to the internal state problem, as had the non-subgrouped RTRL network used by Lindsey(7) in his thesis.

4.2.3 Second Order IIR Lowpass Filter Simulation

The subgrouped RTRL network was trained to emulate a lowpass Butterworth filter, as was described in section 3.4.3. The training files were generated calculating the Butterworth filter response to a binary impulse string (0 0 0 0 0 1 0 0 0 0 0 0 0), and to a series of random values between -1 and 1. Training took place in two steps, first training the network using the random number Butterworth response, and then continuing training on the impulse string training file. This was done because the network appeared to "catch on" to emulating the filter response faster with the random value training file, perhaps due to the richer source of input data to associate with the desired output.

4.2.3.1 Network Impulse Response

The impulse response and frequency response of the network is shown in Figure 22. The frequency response was plotted by performing a fast Fourier transformation (FFT) of the desired network response, and of the network's trained response to a binary impulse.

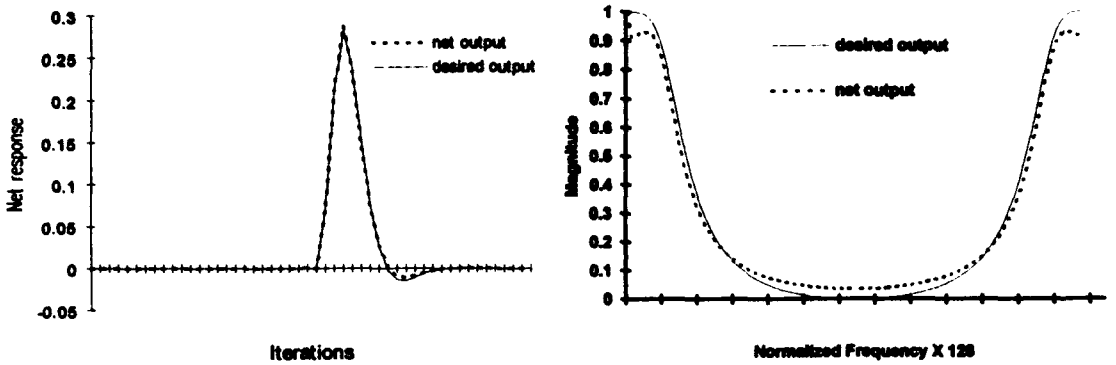


Figure 22: Impulse response and frequency response of the subgrouped RTRL network after training as a Butterworth filter

The impulse frequency response of the trained network matches the desired frequency response well, except for deviations at both the high and low frequencies. This match is closer than was observed by Lindsey, which may be due to several factors. First, Lindsey's training file provided the net with the impulse at the first iteration, while the training file used for the subgrouped RTRL network placed the impulse at t=50. Also, the training data for the subgrouped RTRL network was continuous, i.e. the delayed output from the last iteration was placed as the desired output at t=0. The network outputs were not zeroed at the end of each epoch (see section 4.1.8), removing the discontinuity at t=0. This eliminated the spurious impulse response discussed in section 4.1.8.

4.2.3.2 Unit Step Response

After training the network in emulating the Butterworth impulse response, it was tested on file containing a step function (0 0 0 0 1 1 1 1 1 1), with the IIR filter response as the desired output data. Figure 23 shows the network output versus the desired

Butterworth filter response. The subgrouped RTRL network did not match the overshoot, nor did the final steady state output value match that of the desired output.

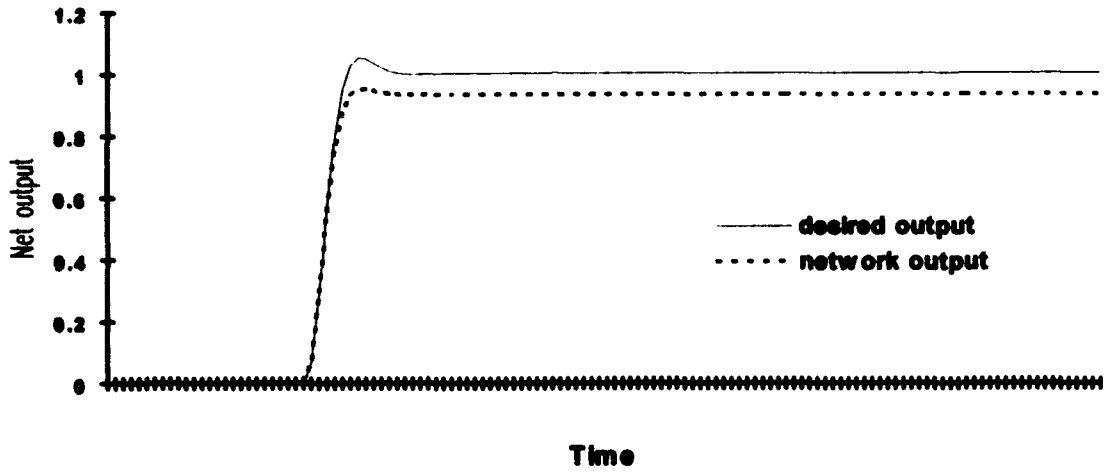


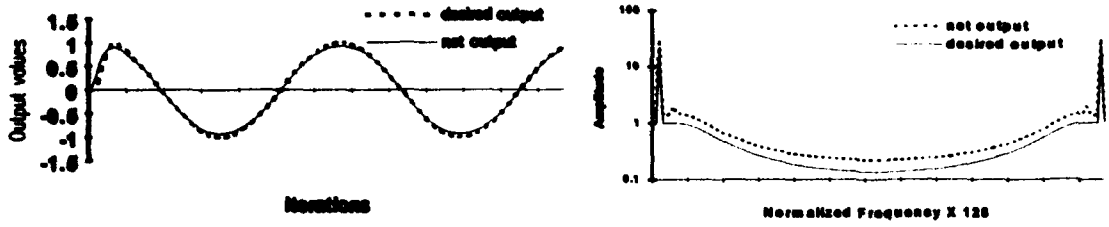
Figure 23: Plot of the Butterworth filter's response to a unit step function versus the subgroup RTRL network's response. Note the lack of overshoot and the lower steady state output of the network.

The lack of overshoot indicates that the RTRL filter is slightly overdamped in its response, while the lower steady state output shows that the network possesses a DC offset after transitioning to the higher state. This DC offset may be caused to some extent by training the network using continous recurrent outputs between epochs (section 4.1.6). When the network outputs are zeroed at the transition between training epochs, a positive DC bias of approximately 0.007 appears at the network output whenever the value should be approaching zero. Making the epochs continuous appears to nearly eliminate this bias. The lower steady state output level for the unit step response function may however be a byproduct of removing the DC bias during training.

4.2.3.3 Sinusoidal Response

The IIR Butterworth filter trained network was tested using a sinusoidal signal as the input, coupled with the Butterworth filtered response as the desired output. As in Lindsey's(7) thesis, the sinusoid consisted of two cycles of a cosine wave divided into 128

sample points. The subgrouped RTRL network closely matched the desired filter response, as can be seen in Figure 24.



(a) (b)

Figure 24: The subgrouped RTRL network closely matched the Butterworth filtered response to a cosine input sinusoid. The frequency response plot is log-linear.

The frequency domain representations of the network output and Butterworth filter response matched rather closely, indicating that the RTRL network was indeed emulating the Butterworth filter for sinusoidal inputs.

4.2.3.4 Pseudo-Random Number Sequence Response

The final test of the subgrouped RTRL algorithm's ability to emulate a Butterworth IIR filter was in the form of the network matching the IIR filtered response to a series of random values, ranging from -1 to 1. The series of random values generated for this test could be interpreted as representing the sampling of a broad spectrum noise signal source. The impulse response trained RTRL network was tested using the random values as the net input, with the Butterworth algorithm filtered response (delayed 1 time step) provided as the desired output.

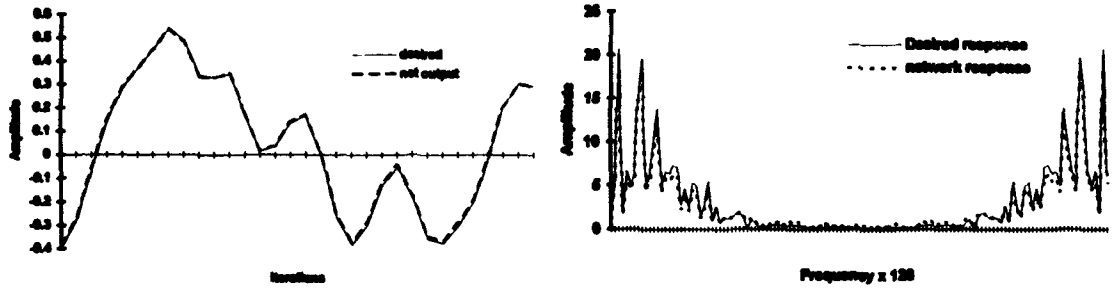


Figure 25: a. A segment of the Butterworth filtered random noise signal data, with the subgrouped RTRL network's output. b. A comparison of the desired frequency response to a noisy (random) signal, versus the subgrouped RTRL output

Figure 25a displays one segment of the filtered signal data with the network's output. The net was able to very closely match the filtered signal, almost to the point of being indistinguishable from the desired signal used as a reference.

A comparison of the spectral characteristics of the filtered noisy signal versus the output generated by the network (Figure25b) reveals that the network closely matched the desired frequency response. The close agreement, throughout the spectrum evaluated, explains why the network output was able to follow the desired filtered response to the noisy signal so accurately. The degree of similarity between the two signals may be due in part to the stage in training the neural network that was performed using a random noise signal as input prior to training on the impulse response.

4.2.3.5 RTRL Versus Subgrouped RTRL Performance

The previously described tests demonstrate the comparable capabilities of the subgrouped RTRL network and the original RTRL algorithm described in Lindsey's thesis(7). While parameters may be changed to enhance the learning accuracy of the network, with the exception of the skipping of the weight update they have no effect on how fast the network learns. The question therefore is, what does subgrouping the network gain us?

This question was answered by comparing the time required to process 10 training epochs by both algorithms, Lindsey's(7) RTRL program and the subgrouped RTRL network. The number of training epochs was chosen to be a small number due to the processing time required by the RTRL network. Longer training runs would show a slight proportional difference in the time required by the subgrouped RTRL network employing weight update skipping, as the percentage of data points skipped varies over time.

To make an honest comparison, the original RTRL code was modified to provide it with the minimum sigmoidal derivative function, which has a major effect on network classification performance. Both networks were set with the minimum sigmoidal derivative factor at 0.01. Each algorithm was tested while varying the number of hidden nodes, to determine the effect on training time. The networks consisted of 20 inputs, six sigmoidal outputs, and 0, 6, 12, 18, 24 and 30 hidden nodes. Training data was a single voice data file derived from the TIMIT voice database and preprocessed using the Payton(8) auditory system algorithm. The subgrouped algorithm was tested under two conditions, with weight update skipping disabled, and with the weight update error threshold set at 0.00001.

Figure 26 shows how the time required to process the 10 training epochs varied between the different algorithms and with varying numbers of hidden nodes. Although only six data points each are shown for the different test runs, it can be clearly seen that the original RTRL code takes much longer to process multi-output problems, and the difference increases geometrically as the number of hidden nodes increase.

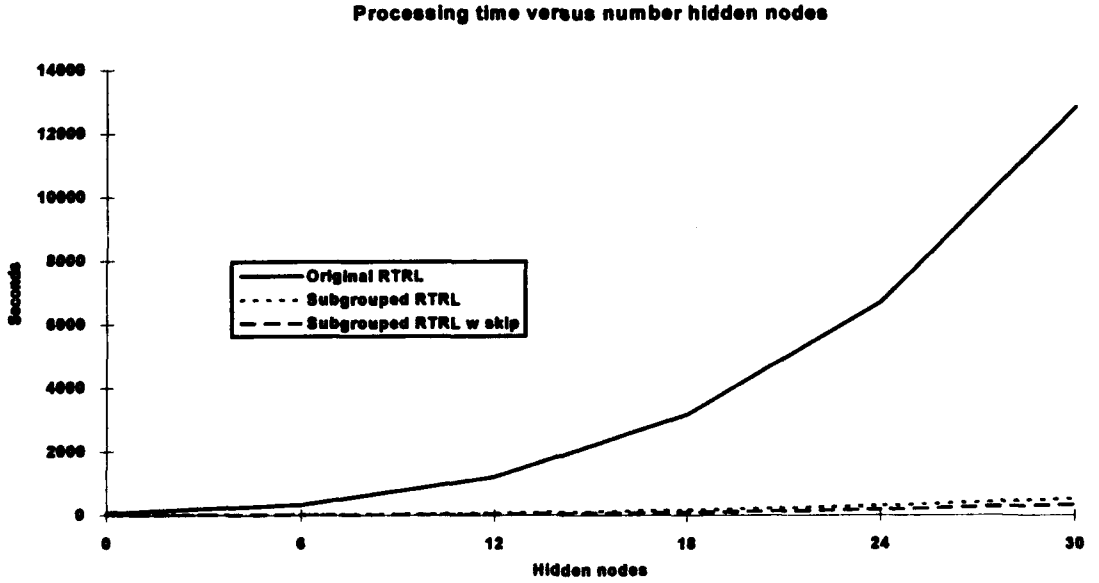


Figure 26: Comparison of network training time between original RTRL code, subgrouped RTRL code, and subgrouped RTRL with weight update skipping enabled.

Figure 27 shows the increase in processing speed obtained when using the subgrouped RTRL algorithm. The speedup was calculated by dividing the time required by the subgrouped RTRL networks into the time required by the original code. The net not utilizing weight update skipping shows a relatively linear speedup when compared to the number of network nodes, showing an approximately O(n) speedup caused by the subgrouping. This is not true of the net that employed weight update skipping, as the speedup is not constant across the different number of hidden nodes. As more hidden nodes are added the improvement for the net employing weight update skipping appears to level off as the network become larger. As the net becomes more complex the average error per iteration rises, and the weight update skipping occurs with less frequency.

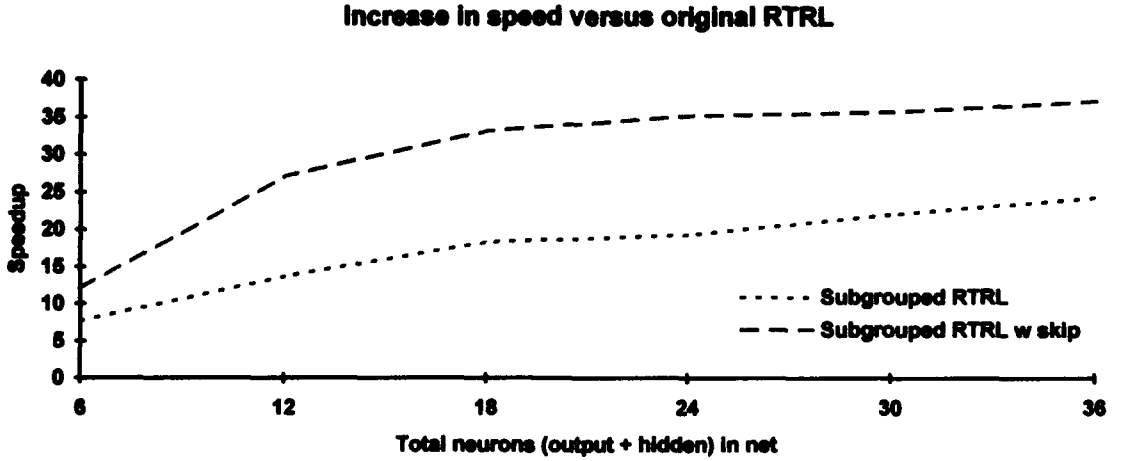


Figure 27: Increase in processing speed of the subgrouped RTRL networks (with and without weight update skipping), versus the original RTRL algorithm

If time to process is not the critical issue, then network accuracy must be examined as well. Figure 28 shows how the accuracy reported by the three network training runs differed over 200 epochs, when each network used 6 hidden nodes. The subgrouped RTRL algorithms performed with lower accuracy than the original non-subgrouped algorithm, while employing the same number of hidden nodes.

The original RTRL code reported a higher average accuracy over the two hundred training epochs, indicating that the subgrouping does incur some reduction in network

capability. This validates Zipser's(20) observation that subgrouping the net can reduce the net accuracy.

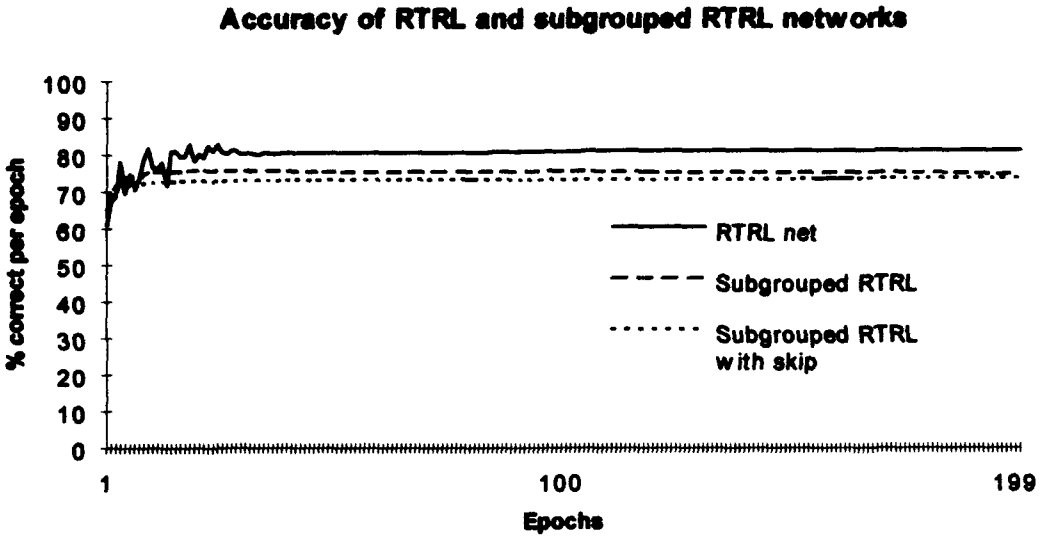


Figure 28: Comparison of the accuracy reported by the original RTRL algorithm, the subgrouped RTRL algorithm, and the subgrouped RTRL algorithm with weight update skipping enabled.

4.3 Network Applications

In section 4.1 of this thesis the effects of varying the network parameters was examined, using the broad class phoneme problem to baseline their impact for that application. Section 4.2 compared the performance of the subgrouped RTRL algorithm with the original RTRL, to examine what the network lost (or gained) in speed, accuracy and capability when it was subgrouped. In this section, the subgrouped RTRL network was applied to two time dependent problems: predicting future behavior based on behavior in the past, and classification based on sequences of feature changes over time.

4.3.1 London Exchange Prediction

The configuration of the network for this application was one input, one linear output, and three sigmoidal hidden nodes. The input to the network, one year's worth of opening market values for the pound in the London Exchange, was paired with the same

data shifted one day ahead in time. This was to train the network to predict what the next day's opening quote would be. Training was initiated with a learning rate of 0.0001, and was competed after 500 epochs. The network was then tested using the opening market values for the pound for a different year, to determine whether the net would match the desired next day values.

The output of the network is shown in Figure 29, along with the desired output. Examination of the figure shows that the network consistently lags behind the desired output. The match, although close, does not demonstrate the net being able to predict the next day's opening quote. Although an enticing possibility, the RTRL algorithm apparently can not be used as a means of predicting changes in the value of the pound using past performance as training data.

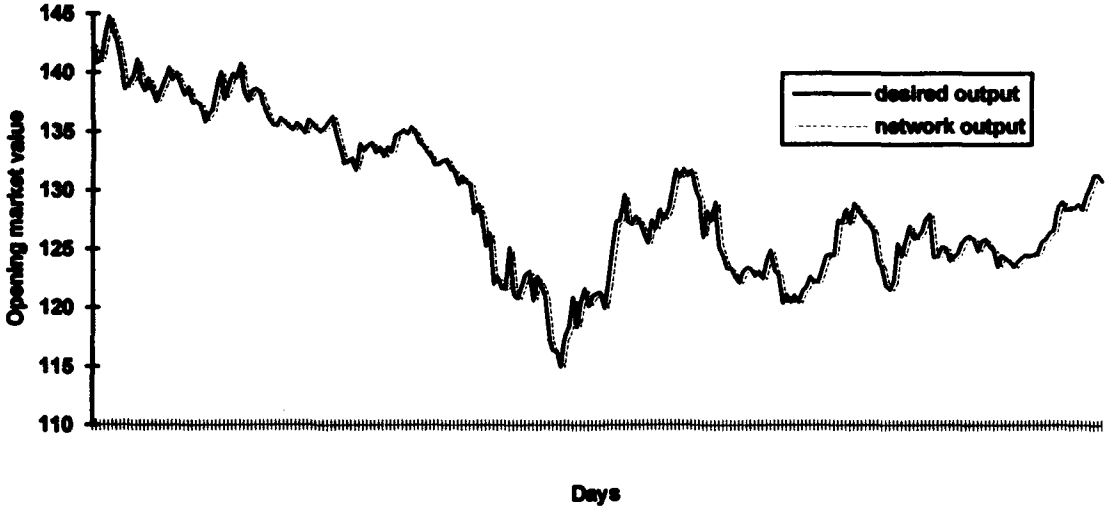


Figure 29: Net performance on test data for London exchange rate prediction

4.3.2 Vehicle Image Classification

The application of the subgrouped RTRL network to the task of classifying vehicles required some repeated attempts before the correct approach was determined. Initially, the net was trained using the sequences of codewords as a single input, providing the net with a "signal" that was hoped would be characteristic of each vehicle.

The net consisted of that single input, plus six sigmoidal hidden nodes and six sigmoidal output neurons. Five of the sigmoidal outputs represented a class of vehicle, one output per type. The sixth output was used to identify the strings of -1 values used to separate the vehicle sequences. Training was initiated using a learning rate of 0.1, and the net was trained for 200 epochs. The net trained very poorly on the sequence information, and so the attempt was repeated using teacher forced learning.

Adding this function to the net training approach appeared to have an immediate and positive effect on the network's ability to differentiate between the vehicle sequences. The net reported a score of +90% within five epochs, and finished after 20 epochs with a scoring of 96.7%. The test file scored similarly, with a 97% accuracy rate. This figure does not mean that the net recognized 97% of the sequences. Instead, this means that the net correctly categorized that percentage of the data points in the file, with each sequence consisting of 14 - 20 data points, and the header spacing between the sequences containing six -1 values.

Because of the rapid training and high accuracy, the code for the subgrouped RTRL was re-examined to verify its startling performance was valid for this task. It was found that in the subroutine in which the desired outputs were substituted for the recurrent network outputs (teacher forced learning), the code did not differentiate between the training and testing of the network. In other words, when the net was being tested with the teacher forced learning selected the substitution was still occurring; the net was "cheating" by looking at the answers during test. When this was corrected, the test score for this task changed to a 46% accuracy.

To resolve this problem, a different approach had to be taken. The net was apparently not able to discern each of the codewords as a "state." Instead, the net had been trained much as it would have been on a analog signal, making codewords adjacent in state nearly equivalent in value for determining a response. To help the net differentiate between the codewords as distinct "states," the input values were converted

into binary code. The range of input values had been from -1 (header value) to 63. To convert this to binary information each input was incremented by two, and then expressed in binary (0000001 to 1000001).

The recurrent network at this point consisted of 7 inputs (binary representation of codewords plus header), six sigmoidal outputs, and 12 hidden nodes. Ten networks were trained over 400 epochs, using an initial learning rate of 0.01 and a momentum of 0.98. After training, the net reported an average 89.7% accuracy rate in recognizing the data points in the training file. The trained networks were then tested, using the 10% of the data source file reserved for this purpose. The nets reported an average of recognizing 89.9% of the test data points. Figure 30 shows how the recognized data points translate into identified sequences.

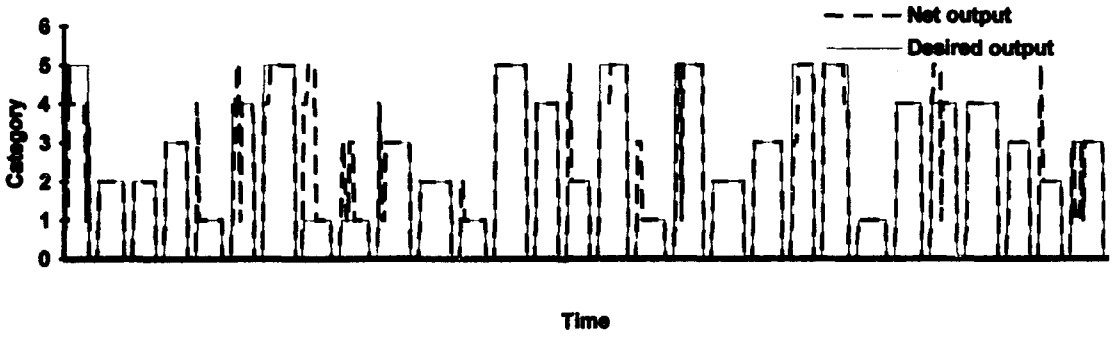


Figure 30: Response of subgrouped RTRL network for sequence test file versus the desired categorical output

Many of the sequences were identified immediately while others experienced some transients, usually at the beginning of the sequence, during which the net misclassified those data points. Because of this, the first data points in the sequences were ignored when determining the vehicle selected by the network. If the class most frequently provided by the net in the last seven points of each sequence is used to classify it, the trained network with the highest accuracy correctly identified almost all (99.22%) of the test sequences. Average accuracy was 96.13%, with a standard deviation of 2.80.

It was surprising how quickly the net selected the correct vehicle in many of the sequences, implying that independent of sequence length, the information necessary to identify the vehicle is often found within the first two or three values of each sequence.

4.4 Summary

The subgrouped RTRL net was tested both to determine the impact of the pararmeters added to enhance performance, and to determine the capabilities and limitations of the network in solving time dependent problems. Each parameter (momentum, minimum sigmoidal derivative factor, weight update skipping, continuity of recurrence between epochs, and maximum sigmoidal input) was varied using the broad class phoneme problem, and the impact of the modification was evaluated.

The effects of the network parameters varied in impact, with the biggest improvement gained by setting a minimum value for the sigmoidal derivative factor in the weight updates. Momentum and the initial learning rate each impacted performance to a lesser extent, and the best values for these parameters are problem dependent, found through trial and error. Weight update skipping provided enough acceleration in network training time that it more than compensated for the small fluctuations in accuracy it caused, and teacher forced learning either did not help net accuracy or dramatically decreased accuracy when the net was tested. Removal of the discontinuity in data and recurrent outputs between training epochs eliminated a spurious impulse response observed when training the net to emulate a low pass filter.

The subgrouped RTRL was also tested to determine whether it was functionally equivalent in performance and characteristics to the RTRL algorithm evaluated in Capt Randall Lindsey's thesis(7). The net was tested on the XOR problem, the internal state problem and the Butterworth filter emulation problems that were discussed in Lindsey's thesis. For each problem, the subgrouped RTRL network performs as well or better than the original algorithm.

For the XOR problem, the subgrouped RTRL net performed similarly in behavior and accuracy to the RTRL network as described in Lindsey's thesis, exhibiting the same temporal dependence in it's selection of valid and invalid XOR inputs, with the network misses scattered across the problem space. Also as in Lindsey's thesis, the subgrouped RTRL network solved the internal state problem with 100% accuracy. For the Butterworth filter problem, the subgrouped RTRL net matched the required output more closely that the RTRL network, which is attributed to the removal of the discontinuity in the impulse response training data and in the recurrent network outputs between training epochs.

Both forms of RTRL networks were also applied to the problem of determining broad phoneme class categories for a single voice file, to quantify differences in training time and accuracy. The number of hidden nodes used by each network was varied during the training trials, to plot net size against training time. Because the subgrouped RTRL algorithm could be accelerated by using weight update skipping it was tested twice, once with weight update skipping disabled and again with the error threshold for skipping weight updates set at a low (0.0001) level.

The subgrouped RTRL net performed significantly better than the original RTRL net in the time required to process the training data (a 7 - 37 times increase in training speed), but it appears subgrouping does cause a tradeoff (8% decrease) in network accuracy. There was also an additional slight tradeoff in network accuracy (1%) for a reduced processing time when the subgrouped RTRL net trained with skipping enabled.

After characterizing the performance of the subgrouped RTRL network, it was applied to two problems: stock market value prediction and vehicle image recognition. The network was able to match the predicted value it was trained to produce relatively well, but the net output consistently lagged the desired predicted value. Because of this, the subgrouped RTRL algorithm would not make a useful tool of any stock analyst if trained in the same manner.

The application of the subgrouped RTRL network performed very well in identifying the five different types of vehicles, based on the sequence of image features provided. The net was only successful in learning this task after the correct format for the input data was applied, i.e. the inputs were expressed in binary to allow the net to differentiate between each codeword as a seperate and distinct state.

V. Conclusions and Recommendations

This thesis represents an effort to improve on the functionality and speed of the RTRL algorithm documented in Capt Randall Lindsey's Master's thesis(7). This effort was performed because of the wide applicability of a time dependent neural network to technical problems facing the Air Force today.

5.1 Conclusions

The subgrouped RTRL algorithm has been demonstrated to be able to solve multiple time dependent problems. Chapter IV details how several of the network parameters enhanced performance in network accuracy and/or time required to process training data. The network was able to solve problems identical or similar to those that were solvable with the original RTRL algorithm, so it appears that subgrouping does not reduce the functionality of the network. These problems (XOR, internal state, second order IIR Butterworth filter simulation) demonstrated the functional equivalence of the two algorithms. It was also demonstrated, using the broad class phoneme identification problem, that the subgrouped RTRL trained in significantly less time, but with less accuracy than the original RTRL network.

The subgrouped RTRL algorithm was applied to two problems: stock market opening value prediction and vehicle image identification. While closely approximating the predicted value of the stock market, the net lagged behind the market behavior enough to make it unwise to use it as a prediction tool. The net performed very well in identifying vehicle images based on time varying image features when the problem was presented properly.

5.2 Recommendations

Based on the results of comparing the two networks, it is recommended that of the two forms of RTRL networks, the subgrouped RTRL network be applied to temporally dependent problems first. If the net fails to provide the required accuracy for the task, then the RTRL

network should be tried. Other avenues for speeding up the RTRL network should also be explored, such as locking those p matrix values that do not change over time, so that the net does not waste training time updating them. It may be possible to start training with a large, multiple hidden node network, and gradually cull out the weights that remain sufficiently small. From a programming perspective, this would be less complex to achieve than incrementally enlarging the net from a smaller configuration to improve accuracy.

On evaluating the differences in performance between the identification of vehicle classes and broad phoneme classes, it might be beneficial to employ similar processing techniques on the voice data to those used to process the vehicle images. The image data was Fast Fourier Transformed (similar in function to the Payton(8) process) and then vector quantized using a clustering algorithm. It may greatly improve the subgrouped RTRL's performance to use a clustering algorithm on the Payton processed voice data, and use the cluster coordinates (or representative codewords) for training the RTRL net. The network would then be using the information embedded in the sequence of data provided to learn to differentiate phonemes, and possibly not from the data points themselves.

5.3 Future Research

It is apparent from the testing of the subgrouped RTRL network that information for solving complex problems may be found not only in features found at each point in time, but also in how the features change over time. The impact of temporally changing information on classification and recognition problems needs to be further explored. Many problems being attacked at this time from a static viewpoint may become more solvable if the added dimension of time is used, particularly in the area of feature recognition. Perhaps time varying features found in aerial views, or in moving faces, may hold the clue for rapid identification.

Appendix A. *Software Development*

The C code for the subgrouped RTRL network is found in Appendix B, along with associated files required for it's compilation and operation. The name of the neural network file used for this thesis is called "recnet.c." The ANSI C code has been run, with minor modifications, on Sun workstations, NeXT workstations, and on a 486 processor IBM compatible PC.

The format for running recnet is "recnet [datafile] [t]". Datafile represents the name of the file containing the network training or test data. If not provided, the net will look for a file named "data.dat" for training data. If "t" (or any added third term) is included with the file name, the net uses the datafile as a test file, based on the weight values stored in "weights.dat."

A.1 File Parameters

At initialization, recnet requires a parameter file named parameter.dat (parametr.dat on PCs) to load in the operating parameters it will train or test under. The following represents the parameters used for most of the tests described in this thesis:

epochs	alpha	seed	moment	y_pr min
100	0.01	152367	0.0	0.01
weights	linear	teacher	skip	cat	loop_data
0	0	0	0.0000	1	0
verbose	max_val	bp_factor			
1	50	0.00			
keep_sum	O	eshold	preview		
0.000	0.		0		

The epochs value determines the number of training epochs the network will run. The learning coefficient, alpha, is set at the beginning of the training run but is halved when the error rate does not change or when the error reported climbs more than a set threshold as the network trains. The seed value is used to initialize the random number

generator, used to create the initial weight values. Moment refers to the momentum factor, while y_pr min is the minimum sigmoidal derivative factor set for the output neurons.

Weights is a flag set to 1 if the net is to continue using the weights found in the file "weights.dat," while a value of one tells the net to create net weight values. Linear is another flag, in which 1 tells the net to output tha activation values of the output neurons and 0 causes the net to provide a sigmoidal output. The teacher flag is set to 1 to enable teacher forced learning, 0 to disable the function. A 1 set for the double flag causes the network to pass through the training data twice during each epoch, with weight updates disabled during the second run. A 0 disables this feature. Skip sets the error threshold during each iteration; above the threshold the net performs weight updates, below the threshold the updates are skipped. Cat set with a value of one tells the net to score the outputs as categories, selecting the output with the highest activation value. A zero on this flag makes the net score each output as good or bad based on whether the output error exceeds the threshold given in OK_threshold. The loop_data flag causes the net to not zero the net output values and p matrix at the end of each epoch when enabled.

The verbose flag enables (or disables) the net's output of information to the screen, while max_val sets the threshold for the activation value of a neuron above which the sigmoidal output is set at one, while a value below the negative of this limit causes the neuron to output a zero. Bp_factor sets the amount by which the backprop algorithm added to the net can influence the weight updates, and usually ranges from 0 - 1. Keep_sum give the net the factor by which it multiplies the neural activation values between data iterations, allowing the past neural activity to influence its current output. OK_threshold is the error threshold for the output neurons, whenever the categorical scoring flag is off. If the error at an output node is within the threshold, it is considered good. Preview is a flag that allows training on the first 25% of the data file, during the first 25% of the training epochs. If the data is uniformly distributed, the net can quickly

generalize on the first 25% and train on the full file for the remaining 75% of the training epochs.

A.2 Output

Recnet will create various files when run, depending on the function selected. These files, and the conditions that cause them to be created are:

Training the network:

weights.dat - save the values of the weight matrix when training is concluded

netout.dat - generated at the end of training, this file contains the network outputs generated during the last epoch with the desired values in a format that will allow training a network based on the net outputs and the desired outputs.

netout2.dat - same as netout.dat, except the activation values of the network are paired with the desired output values.

sequence.dat - created at the end of training when the categorical output function is enabled. Pairs the winning network output with the desired output so that net accuracy can be determined.

Testing the network:

tstcheck.dat - pairs network outputs with desired outputs

testdes.dat - creates a file of the network's desired training values, against which the network output was scored during test

error_tst.dat - provides the net's cumulative error and score as the net passes through the test data

sequence.dat - same purpose as in training, except compares net output with test desired categorical output

Appendix B: *Recurrent Neural Network Code*

This appendix contains a listing of the subgrouped real time recurrent learning source code and its associated files. The files "nrutil.c" and "ran1.c" were derived from the *Numerical Recipes in C* book (11).

```
/* 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:  7 Mar 94

     written by:  Randall L. Lindsey, GEO-91D
     modified by: Jeffrey S. Dean, PTS-92D
***************************************************************************
*/

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

/***************************************************************************
ROUTINE NAME: main
DESCRIPTION: Based on the number of arguments presented when recnet
        is invoked, main causes the net to:
        a. Train on file data.dat
        b. Train on the filename following recnet
        c. Test the accuracy of trained network on the filename
          data

INPUTS: argc - count of arguments following recnet when initiated
         argv - array of argument strings given to recnet

FUNCTIONS CALLED:
check_file() - determines if datafile exists
init_net()  - initializes the network.  Allocates memory for vectors
             and matrices, and initializes them to zero.  Sets
```

```
        random weight values.
read_data() - reads the data from the input file, which include the
        input vectors and training outputs.
read_weights() - reads weights from prior training session, to continue
        training from that point when the weights were saved.
train_net() - trains net based on inputs and training data.

CALLED BY: None
LAST UPDATED: 19 May 1993          BY: Jeffrey S. Dean
*************************************************************************/
main(argc,argv)
int argc;
char *argv[];
{
 switch (argc) {
 case 1:                /* selected if user types "recnet" at prompt.
                           Trains network using data in "data.dat".  */
     datafile="data.dat"; /* Default name of datafile.            */
     check_file();      /* Check to see if the datafile name exists.  */
     init_net(1);        /* 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. */
     if(weights==1)
       read_weights();  /* Read old weights, if restarting learning  */
     train_net();       /* Propagate inputs and update weights based on
                           gradient descent.                    */
     break;

 case 2:                /* selected if user types "recnet <filename>"
                           at prompt. Trains network using <filename>
                           data.                                */
     datafile=argv[1]; /* User specified name of datafile.      */
     check_file();      /* Check to see if the datafile name exists.  */
     init_net(1);        /* 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. */

     if(weights==1)
       read_weights(); /* Read old weights, if restarting learning  */
```

```
        train_net();        /* Propagate inputs, compute outputs, and
                               update weights based on gradient descent.  */
        break;

    case 3:                 /* selected if user types "recnet <filename> t"
                               at prompt. Tests network using <filename>
                               data.                                      */
        datafile=argv[1]; /* User specified name of datafile.          */
        check_file();       /* Check to see if the datafile name exists.  */
        init_net(2);        /* 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_weights(); /* Read weight matrix and saved p states.     */
        read_data();        /* Read data vector array and desired output. */
        test_net();         /* Propagate inputs and compute outputs.      */

        break;

    default:
        printf("\nUsage: recnet [datafilename.dat] [testflag]\n\n");
        break;
    }
    return 0;
} /*  End MAIN() of NET.C  */

/*****************************************************************************
ROUTINE NAME: train_net()
DESCRIPTION: Trains the RTRL net over the selected number of epochs.  The
        user has several options, selected in the "parameters.dat"
        file.  He can:
          - Set the error level above which the net updates its weights.
            Skipping weight updates for accurate outputs can speed learning.
          - Suppress stdout output of net status.  Helps in running
            net in background through automatic backup of host.
          - If output of net represents category membership (1 = member)
            error output of net gives error/times category valid.
          - Have the net "preview" the training data by training on first 25% of the data
            during the first 25% of the training epochs.  Training data must be
homogeneous,
            i.e. the distribution of outputs classes must be spread throughout the data.
INPUTS: None
FUNCTIONS CALLED: net_loop() - Passes data through loop, determines error
            update()  - Updates weight matrix
```

```
            reset_p() - Zeros out p matrix, output vector
            save_weights() - Saves weights of network, plus the outputs
                  (activation function and sigmoid) of the net for
                  one pass through the data
CALLED BY: main()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*************************************************************************/

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

  int numvectors;
  float  climb;
  float  min_error;
  ofp=fopen("error.dat", "w");
  fprintf(ofp,"Total error and percent correct per epoch:\n");
  fprintf(ofp,"Epoch\terror\t\tpercent correct\n");

  numvectors = num_vectors;          /* Set temp variable = number of data vectors */
  min_error = 0.;
  J[1] = J[0] = 0.;
  for(a=0;a<epochs;a++) {
    if(preview==1&&a<(float)epochs*.25) num_vectors = numvectors*.25;
    else num_vectors = numvectors;/* If preview selected, 1st 25% of epochs train on
                                     first 25% of training data.  */
   net_loop(1);                      /* Pass inputs through net, determine error */
   reset_p();                        /*  Zero p_old[][][] matrix for next epoch.   */

   if(verbose==1) {                          /*  If stdout output desired */
    printf("\n%d\t%s %f\t",a,"total error =",J[1]);
    printf("%% correct = %5.2f\t",(float)good/(float)(num_vectors)*100);
    printf("Skipped %5.2f %%\n",(float)skip/(float)num_vectors*100);
   }
   fprintf(ofp,"%d\t%f\t%f\n",a,J[1],(float)good/(float)num_vectors*100);

   if(a==0)
     min_error = J[1];          /* Capture lowest output error */
   min_error = min_error < J[1] ? min_error : J[1];
   climb = J[1] - min_error;
   if(a>3)
     if(climb>0.01*min_error||climb>10||fabs(J[1]-J[0])<0.0000001) {
       alpha = alpha/10.;
       min_error = J[1];
       printf("alpha = %f\n", alpha);
```

```
    }
  if (J[1]<0.000005||alpha<0.000000001) { /*If total error is less than an arbitrary*/
    save_weights();                       /* fractional value, then exit.*/
    fclose(ofp);
    if(verbose==1)
      printf("Stopped on epoch %d\n",a);
    exit(0);
  }

 }                          /* 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() */

/*************************************************************************
ROUTINE NAME: test_net()
DESCRIPTION:  Tests the network accuracy against the data in a test file.
        Calls save_testfiles() to save test data, the output of the
        net as it passes through the test data, and the desired outputs
INPUTS: none
FUNCTIONS CALLED: Net_loop()
CALLED BY: main()
LAST UPDATED: 7 Mar 94                  BY: Jeffrey S. Dean
*************************************************************************/
void test_net()
{
 /* Begin main loop portion */

  ofp=fopen("tstcheck.dat", "w");    /* Open files to record test */
  efp=fopen("testdes.dat", "w");
  yfp=fopen("error_tst.dat", "w");
  if(cat_out==1)
    ufp=fopen("sequence.dat", "w");
  net_loop(2);                       /* Pass data through the net, determine error */
  fclose(ofp);
  fclose(efp);
  fclose(yfp);
  if(cat_out==1)
    fclose(ufp);
```

```
  if(verbose==1) {
     printf("%f percent correct\n",(float)good/(float)num_vectors*100.);
     printf("File 'testcheck.dat' contains test data.\n");
     printf("File 'testdes.dat' contains desired net output test data.\n");
     printf("File 'error_tst.dat' contains test error data.\n");
  }
  return;

} /* end function test_net() */

/*****************************************************************************
ROUTINE NAME: compute_error()
DESCRIPTION:  Computes the error of the net output versus desired output.
          e[k] - error of output at this point in time
          error_vec[k] - error for output k this iteration
          J[1] - Cumulative error on all outputs this epoch
INPUTS: none
FUNCTIONS CALLED: check_if_good() - determines whether the output of the net
             is close enough to the desired output to be valid
CALLED BY: train_net() and test_net()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
void compute_error()
{

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

     loopk(num_outputs)
       error_vec[k] = e[k] = 0.;
     error = 0.;

     if (t>=td || loop_data==1)
       loopk(num_outputs)  {
         e[k] = d[t][k] - y[k*gsize];
         /* Calculate error per output and overall error this iteration */
         error_vec[k] = 0.5 * e[k] * e[k];
         error += error_vec[k];
       }
     if(a==0&&cat_out==1)
       loopk(num_outputs)          /* If using categories & 1st epoch*/
         out_count[k] += d[t  k];  /* tally up how many times each */
                                   /* category appears */

     J[1] += error;
```

```
    good += check_if_good(t);

    return ;
}

/*****************************************************************************
ROUTINE NAME: propagate()
DESCRIPTION:  Passes net output from iteration t-1 to net inputs for
        iteration t.  If teacher forcing function selected, t-1
        outputs to net input replaced with net desired output at t-1.
        Noise is added to the inputs (level entered in parameters.dat)
        proportional to range of input values.
INPUTS: Flag (train) to determine if net is training (=1) or testing (=2)
FUNCTIONS CALLED: none
CALLED BY: net_loop()
LAST UPDATED: 7 Mar 94                  BY: Jeffrey S. Dean
*****************************************************************************/
void propagate(train)
/*  Computes the state of the net at time t, and initializes the z vector for time t.    */
int train;
{
float max, min, diff;

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

    if(teacher==1&&train==1)      /* if teacher forced learning selected, pass */
      loopk(ngroups)              /* previous desired net outputs to net input */
        z[t][m+k*gsize] = d[t][k];

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

    return;
}

/*****************************************************************************
ROUTINE NAME: compute_output()
DESCRIPTION:  Apply non-linear squashing function (sigmoid) to net output
        and hidden layer nodes, unless linear output selected.  If
        selected, net output nodes receive node summation function
        output.
INPUTS: none
```

```
FUNCTIONS CALLED: sigmoid()
CALLED BY: net_loop()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*************************************************************************/
void compute_output()  /*  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 selected, 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]).  */

      loopk(nrows)
         y[k] = sigmoid(s[k]);          /*  Here, y[k]=y(t+1). */

      if(linear==1)                     /* if linear selected, output is summation */
         loopk(num_outputs)             /* function for output nodes          */
            y[k*gsize] = s[k*gsize];

      return ;
}

/*************************************************************************
ROUTINE NAME: update()
DESCRIPTION: Updates weight matrix. Weights can have noise added to update
         to avoid memorizing the exact data path.
         Variable definitions needed to understand subgrouped RTRL:
         - g1 is an offset to position the algorithm at the beginning
           of each subgroup
         - gsize is the size of any subgroup (1 output + hidden nodes)
         - ngroups is the number of subgroups in the net (= # outputs)
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: train_net()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*************************************************************************/
void update()
{
 /* Compute change of weights at time t. delw is reset to zero at each
    iteration (time step), and p_old is p(t).  */

      /* weight changes in subgroup node i = learning rate*output error* |
      | (change in net output g)/(changes in subgroup node output i       |
      | during t-1)                                                       */
      loopg(ngroups)          /* For each subgroup */
      loopij(gsize,ncols) {   /* Change in weight for each node in */
```

```
                        /* subgroup between node and input   */
        delw[g*gsize+i][j]  += alpha*e[g]*p_old[i][j][g*gsize];
    }

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

    loopk(nrows)
        yprime[k] = y[k]*(1.-y[k]); /* Sigmoid function derivative */

    loopk(num_outputs) {
     g1 = k*gsize;    /* g1 points to output node k, first node in subgroup */
     if(linear==0)    /* yp_min sets lower limit for y_prime if output is
                         sigmoidal. Speeds up training if sigmoid
                         derivative can not equal zero.              */
       yprime[g1] = yp_min<yprime[g1]?yprime[g1]:yp_min;
     else
       yprime[g1] = 1.; /* If output linear, y_prime = 1 */
    }

    loopg(ngroups)          /* For each subgroup in the network */
     loopi(gsize)           /* For each node in the subgroup    */
      loopj(ncols)          /* For each input into the network  */
       loopk(gsize) {       /* loop within subgroup */
         kron = 0.0;
         if (i==k) kron = 1.0; /*If input is neuron i's t-1 value */
                         /*use input in p matrix update */
         g1 = g*gsize;   /* subgroup offset */

         /* Sum the product of the p matrix within this subgroup with
         the weight interconnects between the subgroup in the output
         layer and the t-1 subgroup values in the net input layer */

         sum = 0.;
         loopl(gsize)
          if(teacher!=1||l>0)
            sum += w[k+g1][g1+l+m]*p_old[i][j][l+g1];

         /* Update the p matrix */
         p[i][j][k+g1] = yprime[k+g1]*(sum+kron*z[t][j]);

      } /* 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 ;
}

/*****************************************************************************
ROUTINE NAME: reset_delw_s()
DESCRIPTION: Resets the delta weight matrix. Can be set to zero, or can
             retain some of the last weight changes as a momentum factor.
             Activation outputs for the output layer nodes can have selected
             portion retained.
INPUTS:      none
FUNCTIONS CALLED: none
CALLED BY: net_loop and save_weights()
LAST UPDATED: 7 Mar 94                BY: Jeffrey S. Dean
*****************************************************************************/
void reset_delw_s()
{

 /* Reset delta weights using momentum term and reset node sum using */
 /* keep_sum term for next calculation. */

     loopij(nrows,ncols)          /* delta weights multiplied by */
        delw[i][j] *= momentum;   /* momentum factor          */

     loopi(nrows)                 /* Allows use of a kind of activation  */
       s[i] *= keep_sum;          /* function momentum, or a neuron      */
     return;                      /* stimulus that decays over time      */
}

/*****************************************************************************
ROUTINE NAME: reset_p()
DESCRIPTION: Reinitializes old p matrix and output layer node values.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: train_net()
LAST UPDATED: 7 Mar 94                BY: Jeffrey S. Dean
*****************************************************************************/
```

```
void reset_p()
{

 /* Zero p_old[][][] for next calculation. */
     if(loop_data==0) {
      loopg(gsize)
       loopj(ncols)
        loopk(nrows)
           p_old[g][j][k] = 0.;

      loopi(nrows)
        y[i] = 0.;
     }
     return;
}

/*****************************************************************************
ROUTINE NAME: sigmoid()
DESCRIPTION:  Provides sigmoidal squashing function
INPUTS: single precision floating point number
FUNCTIONS CALLED: none
CALLED BY: compute_output()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
float sigmoid(x)
float x;
{
     if (x > max_val)
       return 1.0;
     if (x < -max_val)
       return 0.0;
     return 1/(1 + exp(-x));

} /* end sigmoid */

/*****************************************************************************
ROUTINE NAME: init_net()
DESCRIPTION:  Reads net operating parameters from "parameter.dat" file, as
          well as from the data file.
INPUTS: Flag determining whether net will be trained or tested.
FUNCTIONS CALLED:
          fskip_line()- skips line in input file
          ivector() - allocates memory for integer vector
          vector() - allocates memory for floating point vector
          matrix() - allocates memory for floating point matrix
```

```
            matrix3d() - allocates memory for 3-D fp matrix
            ran1() - random number generator
CALLED BY: main()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*************************************************************************/
void init_net(train)
int train;
{

    char junk_response[256];
    int nrows_w;

 /* Read data from the input file "parameters.dat" */
    if((ifp=fopen("parameters.dat", "r"))==NULL)
      printf("Error opening parameter file\n");
    if((fgets(junk_response, 256, ifp))==NULL) {
      printf("Can't get junk line from parameters file\n");
      exit(0);
    }
    fscanf(ifp,"%d %f %d",&epochs,&alpha,&seed);
    fscanf(ifp,"%f %f",&momentum,&yp_min);
    fskip_line(ifp);
    fskip_line(ifp);
    fscanf(ifp,"%d %d %d",&weights,&linear,&teacher);
    fscanf(ifp,"%f %d %d",&skip_threshold,&cat_out,&loop_data);
    fskip_line(ifp);
    fskip_line(ifp);
    fscanf(ifp,"%d %d %f",&verbose,&max_val,&bp_factor);
    fskip_line(ifp);
    fskip_line(ifp);
    fscanf(ifp,"%f %f %d", &keep_sum, &OK_threshold,&preview);
    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 %d",&num_vectors,&td);
    fclose(ifp);
    if(num_nodes%num_outputs!=0) /* Add hidden nodes until each
                                    subgroup has the same amount */
      num_nodes = ((int)(num_nodes/num_outputs) + 1) * num_outputs;

 /* Output operating parameters to stdout, if selected */
    if(verbose==1) {
```

```
    printf("Recurrent neural net parameters:\n");
    printf("Input file\n");
    printf("%s\n", datafile);
    if(train==1) {
      printf("epochs\t\talpha\t\tmomentum\ty_prime_min\n");
      printf("%d\t\t%f\t%f\t%f\n\n",epochs,alpha,momentum,yp_min);
      printf("vectors\t\tskip threshold\tbp_factor\n");
      printf("%d\t\t%f\t%f\n\n",num_vectors,skip_threshold,bp_factor);
      printf("keep_sum\tmax_val\t\tdata_loop\n");
      printf("%f\t%d\t\t", keep_sum, max_val);
      if(loop_data==1) printf("Enabled  \n\n");
      else printf("Disabled\n\n");
    }
    printf("inputs\t\toutputs\t\thidden nodes\ttime delay\n");
    printf("%d\t\t%d\t\t%d\t\t%d\n\n",num_inputs,num_outputs,
        num_nodes-num_outputs, td);
    printf("Weights\t\tOutput\t\tCategories\tPreview\n");
    if(weights==1||train==2) printf("Old\t\t");
    else printf("New\t\t");
    if(linear==1) printf("Linear\t\t");
    else printf("Sigmoid\t\t");
    if(cat_out==1) printf("Yes\t\t");
    else printf("No\t\t");
    if(preview==1) printf("Enabled\n\n");
    else printf("Disabled\n\n");
    if(cat_out==0)
      printf("\nOK_threshold\n%f\n",OK_threshold);
}

if(weights == 1 || train == 2) {
    if((ifp=fopen("weights.dat", "r"))==NULL) {
      printf("Error opening weight file\n");
      exit(0);
    }
    fscanf(ifp,"%d",&nrows_w);
    fclose(ifp);
    if(nrows_w != num_nodes) {
      printf("***Warning! Weights don't match data configuration!***\n");
      printf("***Replacing data configuration to match weights.***\n");
      num_nodes = nrows_w;
    }
}
m = num_inputs + 1;             /* # of external inputs plus bias */
nrows = n = num_nodes;          /* # of rows for weight matrix */
ncols = m+num_nodes;                    /* # of cols for weight matrix */
```

```
    gsize = num_nodes/num_outputs;        /* number of nodes in a subgroup */
    ngroups = num_outputs;

/* Allocate memory for vectors and matrices */
    out_count=ivector(0,nrows-1);  /* number of times a category output
                                      is the supposed to be output */
    error_vec=vector(0,nrows-1);   /* output error for output node */
    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,gsize-1,0,ncols-1,0,nrows-1);     /* dy/dw */
    p_old=matrix3d(0,gsize-1,0,ncols-1,0,nrows-1);        /* dy/dw */
    accuracy=ivector(0,num_outputs-1);


/* 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] = e[i] = s[i] = error_vec[i] = 0.;
      yprime[i] = yp_min;
      loopj(ncols)
        w[i][j] = delw[i][j] = 0.;
    }
    loopg(gsize)
       loopj(ncols)
         loopk(nrows)
           p[g][j][k] = p_old[g][j][k] = 0.;

    loopi(num_outputs)
       accuracy[i] = 0;

/* Initialize weight matrix using pseudo-random numbers */
    idum = -IABS(seed);
    ran1(&idum);
    loopi(nrows)
      loopj(ncols)
```

```
            w[i][j] = (2*ran1(&idum)-1.0);

  /* Initialize first input to 1 (non-external) */
      loopi(num_vectors)
        z[i][0] = 1.;

      return;
}

/*****************************************************************************
ROUTINE NAME: read_data()
DESCRIPTION:  Reads data file specified for training or test.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: main()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
void read_data()
{

  /* Read data file external inputs */

      if((ifp=fopen(datafile, "r"))==NULL) {
          printf("Error opening data file\n");
          exit(0);
        }
      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]);
          if(d[i][j]!=0&&d[i][j]!=1&&cat_out==1) {
            printf("bad (not category) training value!  %f\n",d[i][j]);
            printf("found on line %d\n",i);
            exit(0);
          }
        }
      }
      fclose(ifp);
      return;
}

/*****************************************************************************
ROUTINE NAME: save_weights()
```

```
DESCRIPTION: Saves network weights. Runs network through one more pass on
        data, capturing network outputs and output node activation
        function values.
INPUTS: none
FUNCTIONS CALLED: reset_delw_s, propagate, compute_output
CALLED BY: train_net()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*************************************************************************/
void save_weights()
{
    int out, desired;
    float max;
    ufp=fopen("weights.dat", "w");
    fprintf(ufp,"%d\n",nrows);
    loopj(nrows)
      fprintf(ufp,"%f ",y[j]);
    fprintf(ufp,"\n");
    loopi(nrows)   /* save network weights */
      loopj(ncols)
        fprintf(ufp,"% f  \n",w[i][j]);
    fclose(ufp);

    if(cat_out==1)
         ofp=fopen("sequence.dat", "w");

    /* save input/outputs in recnet input file format */
    /* to allow further processing using net output data */
    efp=fopen("netout2.dat", "w");  /* Saves activation and desired outputs*/
    fprintf(efp,"%d %d ",num_inputs, num_outputs);
    fprintf(efp,"%d %d %d\n",num_nodes, num_vectors,td);

    ufp=fopen("netout.dat", "w");    /* Saves net output versus desired output*/
    fprintf(ufp,"%d %d ",num_inputs, num_outputs);
    fprintf(ufp,"%d %d %d\n",num_nodes, num_vectors,td);

    loopi(num_outputs)
      accuracy[i] = 0;
    desired = old_des = out = old_out = -1;

    for(t=0;t<num_vectors;t++) {    /* Loop network through data again */

      loopj(num_outputs)            /* save output nodes output */
        fprintf(ufp,"%f \t",y[j*gsize]);

      loopj(num_outputs)            /* save desired output */
```

```
      if(cat_out==1) fprintf(ufp,"% d ",(int)d[t][j]);
      else           fprintf(ufp,"%5.3f ",d[t][j]);
    fprintf(ufp,"\n");

    if(cat_out==1) {
       max = -1000.;              /* find out which of the outputs  */
       loopj(num_outputs)         /* has the highest value          */
         if(s[j*gsize] >max) {
           max = s[j*gsize];
           out = j;
         }
       loopj(num_outputs)         /* Determine the correct output   */
            if(d[t][j] == 1.) desired = j;

       fprintf(ofp,"%d\t%d\n", out,desired); /* Save net output/desired ouput */
    }                                        /* to sequence.dat file for scoring */

    loopj(num_outputs)            /* save activation function, desired output */
      fprintf(efp,"%f ",s[j*gsize]);

    loopj(num_outputs)            /*print desired outputs */
       if(cat_out==1) fprintf(efp,"% d ",(int)d[t][j]);
       else           fprintf(efp,"%5.3f ",d[t][j]);
    fprintf(efp,"\n");

    reset_delw_s();

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

if(cat_out==1) {
  fprintf(ofp,"\n\nPercent correct per category:\n");
  loopk(num_outputs)
    fprintf(ofp,"%f ",100.*(float)accuracy[k]/out_count[k]);
    fprintf(ofp,"\n");
}
  fclose(ufp);
  fclose(efp);
  if(cat_out==1)
```

```
        fclose(ofp);

    return;
}

/*****************************************************************************
ROUTINE NAME: read_weights()
DESCRIPTION:  Reads weights for testing network or for additional training
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: main()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
void read_weights()
{
    int nrows_w;
    ifp=fopen("weights.dat", "r");
    fscanf(ifp,"%d ",&nrows_w);
    loopj(nrows)
      fscanf(ifp,"%f ",&y[j]);
    loopi(nrows)              /* load network weights */
      loopj(ncols)
        fscanf(ifp,"%f",&w[i][j]);
    fclose(ifp);
    return;
}

/*****************************************************************************
ROUTINE NAME: check_file()
DESCRIPTION: Determines if data file exists.  If not, program exits.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: main()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
void check_file()
{

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

```
}

/*****************************************************************************
ROUTINE NAME: save_testfiles()
DESCRIPTION: Saves data from network test.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: test_net()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
void save_testfiles()

{
    int desired, out;
    float max;
    /* Output to testcheck.dat, gives inputs, training values
      and outputs of the net */
    loopj(num_outputs){
      if(cat_out==1)        fprintf(ofp,"(% d :",(int)d[t][j]);
      else                  fprintf(ofp,"(% f :",d[t][j]);
      fprintf(ofp," % f ",y[j*gsize]);
    }
    if(error>OK_threshold)  fprintf(ofp,"    *******");
    fprintf(ofp,"\n");


    /* Output to testdes.dat, shows training values */
    loopj(num_outputs)
        if(cat_out==1)      fprintf(efp,"% d ",(int)d[t][j]);
        else                fprintf(efp,"% f ",d[t][j]);
    fprintf(efp,"\n");

    if(t>0)
    fprintf(yfp,"%f\t%f\n",J[1],(float)good/(float)t*100);

    if(cat_out==1) {
        max = -1000.;           /* find out which of the outputs  */
        loopj(num_outputs)      /* has the highest value          */
          if(s[j*gsize] > max) {
            max = s[j*gsize];
            out = j;
          }
        loopj(num_outputs)      /* Determine the correct output  */
            if(d[t][j] == 1.) desired = j;
```

```
            fprintf(ufp,"%d\t\t%d\n",out,desired);

        }

    return;
}

/*****************************************************************************
ROUTINE NAME: check_if_good()
DESCRIPTION:  Determines if net output matches desired output.  If outputs
              represent membership in categories, routine first checks if any
              output category should be valid.
INPUTS:  Integer value representing position in data stream (iteration)
FUNCTIONS CALLED: none
CALLED BY: net_loop()
LAST UPDATED: 7 Mar 94                    BY: Jeffrey S. Dean
*****************************************************************************/
int check_if_good(iter)
int iter;
{
int good_one, out, count;
float max;

      good_one = 0;                   /* initialize flag */
      if((t<td) && (loop_data == 0))
       ++good_one;
      else  {
        if(cat_out==1) {
           max = -1000.;              /* find out which of the outputs  */
           loopj(num_outputs)         /* has the highest value          */
             if(s[j*gsize] >max) {
               max = s[j*gsize];
               out = j;
               if(out<0||out>num_outputs-1) {
                 printf("out = %d\n");
                 exit(0);
               }
             }
           if((int)d[iter][out]==1){          /* If the highest value matches  */
             good_one++;                      /* the desired category, its good.*/
             accuracy[out] += 1;              /* Net has hit in category, inc count*/
           }
        }
        else  {                               /* If the output is not a category */
          count = 0;
```

```
            loopj(num_outputs)
              if(error_vec[j]>OK_threshold)  /* check if the error is low */
                count++;
            if(count==0)
              good_one++;
          }
        }
        if(good_one > 1)
          good_one = 1;
        return good_one;
}

/*****************************************************************************
ROUTINE NAME: net_loop()
DESCRIPTION:  Called for each data point, it computes the error from the
              last iteration, checks whether the output can be considered
              valid, resets the delta weight matrix, passes the output and
              hidden node values from the last iteration to the net input
              layer, and computes the net output for this iteration.
INPUTS: Flag (train) to determine if net is training (=1) or testing (=2)
FUNCTIONS CALLED:
        compute_error() - determines the error between the net
              output from the last iteration and the desired output
        reset_delw_s() - Resets the delta weight matrix and zeros
              out the weighted summed inputs from the last iteration
        propagate() - passes the values produced by the top layer
              of the network (hidden and output nodes) back to the
              net input for this iteration
        compute_output() - computes the values of the output and
              hidden nodes of the net
        save_testfiles() - saves test data, net output & desired
                    output
CALLED BY: train_net(), test_net(), save_weights()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
void net_loop(train)
int train;
{
    J[0] = J[1];          /* Update error from last epoch. */
    J[1] = 0.;            /* Initialize error for current epoch. */
    skip = 0;             /* Initialize skipped updates counter. */
    good = 0;             /* Initialize # right answers counter. */
    for(t=0;t<num_vectors;t++) {
     compute_error();     /* Computes the error at time t.
                             How far off are the outputs from the
```

```
                        desired values? Compute total error.*/

    if(train==2)
     save_testfiles();

    reset_delw_s();

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

    if(train==1) {
      if(error>skip_threshold)       /*If error above threshold, update weights */
        update();                    /*Computes del_w(t), and p(t+1). Backprop */
      else skip++;                   /* error through net and perform gradient */
                                     /*  descent to calculate the delta weights.  */
      if(bp_factor>0.&&t>0)
        loopij(num_outputs,ncols)
          w[i*gsize][j] += alpha*e[i]*bp_factor*yprime[i*gsize]*z[t-1][j];
    }
  }

    return;
}
```

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

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

     date: 30 May 91

     written by: Randall L. Lindsey
*******************************************************************************/
float *vector();
float **matrix();
float ***matrix3d();
float ran1();
int *ivector();
int **imatrix();

FILE *ifp, *ofp, *afp, *efp, *ufp, *yfp;
int run=1;
char str[80], *datafile;
int *out_count;
int nrows, ncols, g, i, j, k, l, m, n,num_categ,output_sel,cat_out;
int epochs, a, b, t, hold=5,inc, weights, norm,teacher,td, verbose;
int num_inputs, num_outputs, num_nodes, num_vectors, dble, reset;
int seed, idum=1,out_fb, linear,gsize,g1,ngroups,data_group,good,bad;
int loop_data, max_val, skip;
float J[2], sum, kron,x,yp_min,momentum,junk;
float alpha, bp_factor,keep_sum;
float alpha1,error,skip_threshold,*latency, *lat_value;
float input_noise, weight_noise, *error_vec;
float *y, *s, *e, *f, *yprime,*y_won,*mean_vect,*vect_max;
float **z, **d, **w, **delw,**y_old,**sum_out;
float ***p, ***p_old, ***p_temp;
int *accuracy;
float sigmoid();
void init_net();
void train_net();
void test_net();
void read_data();
void propagate();
void propagate_t();
void compute_output();
void compute_error();
void update();
void reset_delw_s();
void reset_p();
```

```
void init_weights();
void save_weights();
void read_weights();
void check_file();
void save_testfiles();
int check_if_good();
void net_loop();


/*** MACRO.H ************************************************/

/*#define TRAIN true;*/

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 loopg(A) for(g=0;g<A;g++)
#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))))
#define INT_MAX (2147483647)
/** Dividing by 100 insures that cc and gcc give same results **/
#define IRAN1(A) ((int)(ran1(A)*(float)INT_MAX)/100)
```

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

NRUTIL.C Numerical utility routines; allocate memory for vectors and 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;
}

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;
}



float **submatrix(a,oldrl,oldrh,oldcl,oldch,newrl,newcl)
float **a;
int oldrl,oldrh,oldcl,oldch,newrl,newcl;
{
      int i,j;
      float **m;

      m=(float **) malloc((unsigned) (oldrh-oldrl+1)*sizeof(float*));
```

```
    if (!m) nrerror("allocation failure in submatrix()");
    m -= newrl;

    for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+oldcl-newcl;

    return m;
}



void free_vector(v,nl,nh)
float *v;
int nl,nh;
{
    free((char*) (v+nl));
}

void free_ivector(v,nl,nh)
int *v,nl,nh;
{
    free((char*) (v+nl));
}

void free_dvector(v,nl,nh)
double *v;
int nl,nh;
{
    free((char*) (v+nl));
}

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

    for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
    free((char*) (m+nrl));
}

void free_dmatrix(m,nrl,nrh,ncl,nch)
double **m;
int nrl,nrh,ncl,nch;
{
    int i;
```

```
    for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
    free((char*) (m+nrl));
}

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

    for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
    free((char*) (m+nrl));
}

void free_submatrix(b,nrl,nrh,ncl,nch)
float **b;
int nrl,nrh,ncl,nch;
{
    free((char*) (b+nrl));
}

float **convert_matrix(a,nrl,nrh,ncl,nch)
float *a;
int nrl,nrh,ncl,nch;
{
    int i,j,nrow,ncol;
    float **m;

    nrow=nrh-nrl+1;
    ncol=nch-ncl+1;
    m = (float **) malloc((unsigned) (nrow)*sizeof(float*));
    if (!m) nrerror("allocation failure in convert_matrix()");
    m -= nrl;
    for(i=0,j=nrl;i<=nrow-1;i++,j++) m[j]=a+ncol*i-ncl;
    return m;
}

void free_convert_matrix(b,nrl,nrh,ncl,nch)
float **b;
int nrl,nrh,ncl,nch;
{
    free((char*) (b+nrl));
}
```

```
/********************malloc.h   1.2 ************************/

/*
     Constants defining mallopt operations
*/
#define M_MXFAST        1    /* set size of blocks to be fast */
#define M_NLBLKS        2    /* set number of block in a holding block */
#define M_GRAIN         3    /* set number of sizes mapped to one, for
                                small blocks */
#define M_KEEP          4    /* retain contents of block after a free until
                                another allocation */
/*
     structure filled by
*/
struct mallinfo  {
     int arena;     /* total space in arena */
     int ordblks;   /* number of ordinary blocks */
     int smblks;    /* number of small blocks */
     int hblks;     /* number of holding blocks */
     int hblkhd;    /* space in holding block headers */
     int usmblks;   /* space in small blocks in use */
     int fsmblks;   /* space in free small blocks */
     int uordblks;  /* space in ordinary blocks in use */
     int fordblks;  /* space in free ordinary blocks */
     int keepcost;  /* cost of enabling keep option */
};

char *malloc();
void free();
char *realloc();
int mallopt();
struct mallinfo mallinfo();


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

RAN1.C - Numerical recipes 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



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

MAKEFILE

*********************************************************************/
CFLAGS = -O2  -lm

recnet : recnet.c nrutil.o ran1.o
     cc -o recnet recnet.c nrutil.o ran1.o $(CFLAGS)

nrutil.o : nrutil.c
     cc -O2 -c nrutil.c

ran1.o : ran1.c
     cc -O2 -c ran1.c

clean:
     rm -f *.o net recnet
```

The following listing is from the "parameters.dat" file, used to define the working parameters under which the recurrent net is operating

epochs	alpha	seed	moment	y_pr min
1000	0.01	152367	0.0	0.01
weights	linear	teacher	skip	cat	loop_data
1	0	0	0.0000	1	0
verbose	max_val	bp_factor			
1	50	0.50			
keep_sum	OK_threshold	preview			
0.000	0.125	0			

epochs = number of times net trains on data file
alpha = learning constant
seed = random number seed
moment = momentum term
y_pr min = minimum value allowed for sigmoidal derivative function f(1-f)

weights = 0: generate new weights for this training session
1: used the weights in "weights.dat" to continue training

linear = 0: output nodes use sigmoidal output
1: output nodes use linear output

teacher = 0: do not use teacher forced training
1: use teacher forced training

skip = error threshold above which weights are updated

cat = 0: outputs of net do not represent categories
1: outputs of net represent categories (i.e. are 1 or 0)

loop_data = 0: zero out outputs after end of epoch
1: Do not zero out outputs. Allows continuity of data passing through the net between epochs

verbose = 0: Do not print messages to stdout (screen)
1: Print messages to stdout (screen)

max_val = limit of activation value. Above max_val, the sigmoid function returns 1; below -max_val, the sigmoid function returns 0.

bp_factor = Gives net capability to update weights by means of standard backprop algorithm, in addition to RTRL. Factor determines how much emphasis given to backprop weight updates. Usual range between 0 and 1. Backprop only used on weights to output nodes.

keep_sum = Provides a momentum term for the activation values of the neurons.

preview = 0: Net trains on all the training data, each epoch
1: For 1st 25% of epochs, net trains on 1st 25% of training data. Remaining 75% of epochs training occurs with all training data.

Appendix C: *Source Code for Creation/Manipulation of Data*

This appendix contains listings of the source code for the program used to generate/ modify the data used to train or test the subgrouped recurrent network, or to evaluate network accuracy based on net outputs. Code was added as need occured, so no claim is made as to program efficiency or organization.

```
/*****************************************************************************
CREATE.C

    A tool to allow manipulation of the data files used to train
    and test recnet.

    date: 7 May 93

    written by: Jeffrey S. Dean
*****************************************************************************/
#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

#include <stdio.h>
#include "macros.h"
#include <math.h>
#include "def.h"
#include <string.h>
/*****************************************************************************
ROUTINE NAME: main
DESCRIPTION:  Prompts the user whether he wants to create a file to train
        or test the net on a Butterworth filter response, or to load
        and mainipulate a data file.
INPUTS: default inputs argc and argv, not used
FUNCTIONS CALLED:
 Butterworth() - Prompts user to select type of Butterworth filter data
 File_work() - Prompts user for file name to be loaded, then for function
        to be performed
CALLED BY: none
```

```
LAST UPDATED: 7 May 93                BY: Jeffrey S. Dean
*****************************************************************************/
main(argc, argv)
int argc;
char *argv[];

{
 o = matrix(0,25000,0,64);
 v = matrix(0,25000,0,50);
 pick=matrix(0,25000,0,1);
 pntr=imatrix(0,1500,0,1);
 o2 = matrix(0,25000,0,6);
 v2 = matrix(0,25000,0,28);
 num_vectors = numvectors = 0.;
 Select();
 exit(0);
}
void Select()
{

 int choice;
 for (;;) {
  printf("Choose one of the following: \n");
  printf("\n1. Create a Butterworth filter response file \n");
  printf("\n2. Load and modify an existing file \n");
  printf("\n3. Create a XOR data file \n");
  printf("\n4. Manipulate sequence identification files \n");
  scanf("%d", &choice);
  printf("\n");
  if(choice==1) Butterworth();
  if(choice==2) {
   Append();
   File_work();
  }
  if(choice==3) Xor();
  if(choice==4) Sequence();
  printf("That is not a valid choice\n\n\n");
 }
}
/*****************************************************************************
ROUTINE NAME: Butterworth()
DESCRIPTION: Prompts user to select between cosine, step, random or impulse
        functions for building a Butterworth filter data file for
        recnet.
INPUTS: none
```

```
FUNCTIONS CALLED:
 Cosin() - creates 128 point cosine wave values, with Butterworth filter
        response as training values
 Step()  - creates a step function (0 to 1) input file, with Butterworth
        filter response as training values
 Random() - Creates a 699 point random number string (0 to 1 values), with
        Butterworth filter response as training values
 Impulse() - creates a 200 point series of impulses, with Butterworth filter
        values as training data
CALLED BY: main()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*************************************************************************/
 void Butterworth()
 {
  int choice;
  for (;;) {
     printf("\nDo you want to:\n");
     printf("1. Generate a cosine function training file?\n");
     printf("2. Generate a step function training file?\n");
     printf("3. Generate a random function training file?\n");
     printf("4. Generate an impulse function training file?\n");
     printf("5. Exit\n");
     skip_line;
     scanf("%d", &choice);
     printf("\n");
     if(choice==1)  Cosin();
     if(choice==2)  Step();
     if(choice==3)  Random();
     if(choice==4)  Impulse();
     if(choice==5)  exit(0);
     printf("That is not a valid choice\n\n\n");
  }
 }

/*************************************************************************
ROUTINE NAME: File_work()
DESCRIPTION: Allows the user to select from multiple data file manipulation
        options
INPUTS: none
FUNCTIONS CALLED:
 Append() - Appends another data file to the data in memory.  Number of file
        inputs and outputs must match data format in memory
 Save()  - Saves the current form of data as a file
 Merge() - Allows the user to replace the inputs or outputs of the data in
        memory with those in a data file.  Number of file data vectors must
```

```
        match with the number of data vectors in memory.
Time_delay() - shifts the outputs ahead in time
Categories() - Prompts the user to select an output (integer) and expands
        the output into category format (1 = member, 0 = nonmember)
Cull() - Extracts the data vectors in the data file that belong to one of
        the possible categories
Norm() - Normalizes the inputs
Clear() - Reinitializes inputs and outputs
One_cat() - Specifically for phoneme group extraction.  Performs one of two
        functions: Expands outputs to all categories in a phoneme
        group (nasal, vowel, etc.), with one category for non-members;
        or converts output to two membership functions, either in group
        or not in group.
Differentiate() - Differentiates inputs across each vector and between
        vectors.
Status() - displays number of data vectors, number of inputs/outputs, and
        the time delay in the outputs.
Out_types() - Displays how many of the potential categories are present in
        the data
View()  - Allows user to display current inputs and outputs
Phoneme() - Shows user which phonemes of each of the phoneme types are
        present in the data
Compare() - If user merges outputs of file used to train/test net with
        outputs net produced, the routine checks to see if the net
        provided the right answer, broken down across output categories
CALLED BY: main()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*************************************************************************/
 void File_work()
 {
  int select;
  for (;;) {
    printf("\nDo you want to:\n");
    printf("FILE TRANSFER FUNCTIONS -\n");
    printf("1. Append file\t2. Save file\t3. Merge file\n");
    printf("\nDATA MANIPULATION FUNCTIONS -\n");
    printf("4. Add time delay\t\t5. Expand in/outputs\t6. Cull outputs\n");
    printf("7. Normalize inputs?\t\t8. Clear data\t\t9. Select category\n");
    printf("10. Differentiate inputs\n");
    printf("\nDATA VIEWING FUNCTIONS -\n");
    printf("11. Check status\t\t12. Check outputs\t13. View data\n");
    printf("14. Show phoneme breakdown\t15. Compare inputs/outputs\n");
    printf("\n16. Exit\n");
    skip_line;
    scanf("%d", &select);
```

```
    printf("\n");
    if(select==1)  Append();
    if(select==2)  Save();
    if(select==3)  Merge();
    if(select==4)  Time_delay();
    if(select==5)  Categories();
    if(select==6)  Cull();
    if(select==7)  Norm();
    if(select==8)  Clear();
    if(select==9)  One_cat();
    if(select==10) Differentiate();
    if(select==11)  Status();
    if(select==12)  Out_types();
    if(select==13)  View();
    if(select==14)  Phoneme();
    if(select==15)  Compare();
    if(select==16)  exit(0);
   }
  }


/*****************************************************************************
ROUTINE NAME: Append()
DESCRIPTION:  Prompts user for another file name, to append to the data
          already in memory.  Will not load file if the number of
          inputs or outputs in the file do not match the data in memory.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Append()
 {
  char choice;
  int num_inputs, num_outputs;

  printf("\nWhat is the name of the file? :");
  skip_line;
  scanf("%s", datafile);
  printf("\n");
  printf("Reading %s . . . \n",datafile);
  ifp=fopen(datafile,"r");
  fscanf(ifp,"%d %d %d",&num_inputs,&num_outputs,&numvectors);
  printf("This file has %d inputs, %d outputs, ",num_inputs,num_outputs);
  printf("and %d vectors.\n",numvectors);
```

```
  if(num_inputs != numinputs&&numinputs!= 0) {
    printf("***********CAN NOT CONTINUE!!**********\n");
    printf("number of inputs not the same as before\n");
    return;
  }
  if(num_outputs != numoutputs&&numoutputs!= 0) {
    printf("***********CAN NOT CONTINUE!!**********\n");
    printf("number of outputs not the same as before\n");
    return;
  }
  printf("Continue? (y/n)");
  skip_line;
  scanf("%c", &choice);
  printf("\n");
  if(choice=='y'){
    numinputs = num_inputs;
    numoutputs = num_outputs;
    fskip_line(ifp);
    printf("loading data file ...\n");
    loopi(numvectors) {
       loopj(numinputs)
         fscanf(ifp,"%f",&v[i+num_vectors][j]);
       loopj(numoutputs)
         fscanf(ifp,"%f",&o[i+num_vectors][j]);
    }
  num_vectors += numvectors;
  }
  fclose(ifp);
  return;
 }

/*****************************************************************************
ROUTINE NAME: Time_delay()
DESCRIPTION: Shifts output values in data a user selected number of
        data vectors.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Time_delay()

 {
  float tail[5][50];
```

```
  printf("How many ticks do you want to delay the output?\n");
  skip_line;
  scanf("%d", &td);
  printf("\n");
  printf("numvec=%d, numout=%d, td=%d\n",num_vectors, numoutputs,td);
  if(td>0) {
   loopi(td)
      loopj(numoutputs)
        tail[i][j] = o[num_vectors+i-td][j];
   loopi(num_vectors-td)
      loopj(numoutputs)
        o[num_vectors-i-1][j] = o[num_vectors-i-1-td][j];
   loopi(td)
      loopj(numoutputs)
        o[i][j] = tail[i][j];
   TD += td;
  }
  return;
 }

/*****************************************************************************
ROUTINE NAME: Categories()
DESCRIPTION: Selects one of integer outputs, asks for the range of values
        represented by the output (how many potential categories) and
        expands the output value to a string of 1s and 0s, with 1
        representing membership in a category.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: file_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Categories()
 {
  int index, io, max, rep;
  printf("1. Inputs\n2. Outputs\n");
  skip_line;
  scanf("%d", &io);
  printf("\n");
  printf("How many categories does this break down to?\n");
  skip_line;
  scanf("%d", &cat);
  printf("\n");
  if(io==2) {
     printf("Which output do you want? (1 - %d)\n",numoutputs);
     skip_line;
```

```
    scanf("%d", &sel);
    printf("\n");
    loopi(num_vectors) {
          index = o[i][sel-1];
          loopj(cat)
            o[i][j] = 0.;
          if(index>=0)
            o[i][(int)index] = 1.;
    }
    numoutputs = cat;
}
else if(io==1) {
    printf("1. Binary representation\n2. Fully expanded\n");
    skip_line;
    scanf("%d", &rep);
    printf("\n");
    if(rep==2) {
     loopi(num_vectors) {
           index = v[i][0]+1;
           loopj(cat)
             v[i][j] = 0.;
           if(index>=0)
             v[i][(int)index] = 1.;
           else v[i][0] = 1.;
     }
     numinputs = cat;
    }
    else if(rep==1) {
     loopi(num_vectors) {
          max=64;
          index = v[i][0]+2;
          loopj(7)
            v[i][j] = 0.;
          loopj(7) {
           if(index>=max) {
             index -= max;
             v[i][j] = 1;
           }
           max = max/2;
          }
     }
     numinputs = 7;
    }
}
return;
```

```
 }
/*****************************************************************************
ROUTINE NAME: Cull()
DESCRIPTION: Prompts user to select one of integer outputs, and asks user to
        select one of potential categories represented by this output.
        Culls out those data vectors that do not belong to this category.
        Allows user to include those non-selected category vectors
        just before and immediately after the data vectors selected.
        This routine allows user to extract only vowels or a specific
        phoneme from a voice data file.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Cull()
 {


int out, count, count2;
   char choose_lead, all;
   printf("Which output do you want (1-%d)? ",numoutputs);
   skip_line;
   scanf("%d", &out);
   printf("\n");
   printf("Which category do you want to extract? ");
   skip_line;
   scanf("%d", &cat);
   printf("\n");
   printf("Do you want the vector before the desired category?\n");
   printf("(This will provide a lead in to the desired section)\n");
   skip_line;
   scanf("%c", &choose_lead);
   printf("\n");
   printf("Do you want all the vectors?  ");
   skip_line;
   scanf("%c", &all);
   printf("\n");
   loopi(num_vectors)
      pick[i][0] = 0;
   if(all == 'y')
    loopi(num_vectors)
      pick[i][0] = 1;
   loopi(num_vectors)
      if(o[i][out-1]==cat)
```

```
        pick[i][0] = 1;
    if(choose_lead == 'y')
     loopi(num_vectors) {
      if(i>1&&i<num_vectors-2) {
       if(o[i+1][out-1]==cat||o[i+2][out-1]==cat)
        pick[i][0] = 1;
       if(o[i-1][out-1]==cat||o[i-2][out-1]==cat)
        pick[i][0] = 1;
      }
    }
    count = 0;
    count2 = 0;
    loopi(num_vectors)
      if(pick[i][0] == 1) {
        loopj(numinputs)
            v[count][j] = v[i][j];
        if(o[i][out-1]==cat){
            o[count][0] = 1.;
            o[count][1] = 0.;
            count2++;
        }
        else {
            o[count][0] = 0.;
            o[count][1] = 1.;
        }
        count += 1;
      }
    num_vectors = count - 1;
    numoutputs = 2;
    printf("Number of vectors = %d,",num_vectors);
    printf(" number of desired categories = %d,",count2);
    return;
  }
/*****************************************************************************
ROUTINE NAME: Norm()
DESCRIPTION:  Determines max and min of each data vector, subtracts the
          average of the max and min values to center data on zero.
          Divides each input by half the range of input values to size
          the values between -1 and 1.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                BY: Jeffrey S. Dean
*****************************************************************************/
 void Norm()
```

```
{
float  min, max;

 loopi(num_vectors) {
    min = 100000.;
    loopj(numinputs)
       if(min>v[i][j]) min = v[i][j];
    max = 0.;
    loopj(numinputs)
       if(max<v[i][j]) max = v[i][j];
    loopj(numinputs)
       v[i][j] -= (max+min)/2;
    loopj(numinputs)
       v[i][j] /= (max-min)/2;
 }
 return;
}
/*****************************************************************************
ROUTINE NAME: Save()
DESCRIPTION:  Saves the data as a file
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Save()
 {

  char integer_out, integer_in;
  printf("What do you call the output file?  ");
  skip_line;
  scanf("%s",outfile);
  printf("How many hidden nodes do you want?   ");
  skip_line;
  scanf("%d", &numnodes);
  printf("Are the outputs integers? (y/n)  ");
  skip_line;
  scanf("%c",&integer_out);
  printf("\n");
  printf("Are the inputs integers? (y/n)  ");
  skip_line;
  scanf("%c",&integer_in);
  printf("\n");
  ofp=fopen(outfile,"w");
  printf("\nSaving data .... \n");
```

```
  fprintf(ofp,"%d %d %d ",numinputs,numoutputs,numnodes+numoutputs);
  fprintf(ofp,"%d %d\n",num_vectors, TD);
  loopi(num_vectors) {
    if(integer_in=='y')
      loopj(numinputs)
        fprintf(ofp,"%d ",(int)v[i][j]);
    else
      loopj(numinputs)
        fprintf(ofp,"%12.10f ",v[i][j]);
    if(integer_out=='y')
     loopj(numoutputs)
       fprintf(ofp,"%d ",(int)o[i][j]);
    else
     loopj(numoutputs)
      fprintf(ofp,"%12.10f ",o[i][j]);
    fprintf(ofp,"\n");
  }
  fclose(ofp);
  return;
 }

/*****************************************************************************
ROUTINE NAME: View()
DESCRIPTION: Prints current values of inputs and outputs to screen
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void View()
 {
  char cont;
  int count = 0;

  cont = NULL;
  loopi(num_vectors) {
   loopj(numinputs)
     printf("%4.2f ",v[i][j]);
   printf("\n");
   loopj(numoutputs)
     printf("%4.2f ",o[i][j]);
   printf("\n");
   count += 1;
   if(count >10) {
    printf("Press <return> to continue, q to quit\n");
```

```
      skip_line;
      scanf("%c", &cont);
      printf("\n");
      count = 0;
     }
     if(cont == 'q') break;
   }
   return;
 }

/*****************************************************************************
ROUTINE NAME: ran1()
DESCRIPTION: Random number generator
INPUTS: integer pointer for random number seed
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                    BY: Jeffrey S. Dean
*****************************************************************************/
 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) printf("%s\n","RAN1: This cannot happen.");
    temp=r[j];
    r[j]=(ix1+ix2*RM2)*RM1;
    return temp;
 }
/*****************************************************************************
ROUTINE NAME: Random()
DESCRIPTION: Generates a 699 point random number sequence, with the response
        of a Butterworth filter associated with each point.
INPUTS: none
FUNCTIONS CALLED: ran1(), Save()
CALLED BY: Butterworth()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Random()
 {
    float class,a0,a1,a2,b0,b1;
    int idum=1,i,j,bubba;

    a0=0.0676; a1=0.1352; a2=0.0676;
    b0=1.1422; b1=-0.4124;
    idum = -IABS(737496732);
    ran1(&idum);
    o[0][0]=o[1][0]=0.0;
    loopi(710)
      v[i][0] = o[i][0] = 0.;
    loopi(600)
      v[i+50][0] = 2.0*ran1(&idum)-1.0;
    num_vectors = 700;
    loopj(3) {
     loopi(700)
     o[i+2][0]=a0*v[i+2][0]+a1*v[i+1][0]+a2*v[i][0]+b0*o[i+1][0]+b1*o[i][0];
     o[0][0]=a0*v[0][0]+a1*v[699][0]+
          a2*v[698][0]+b0*o[699][0]+b1*o[698][0];
     o[1][0]=a0*v[1][0]+a1*v[0][0]+
          a2*v[699][0]+b0*o[0][0]+b1*o[699][0];
    }
    numinputs = 1;
    numoutputs = 1;
    Time_delay();
    Save();
    exit(0);
    return;
}
```

```
/*****************************************************************************
ROUTINE NAME: Impulse()
DESCRIPTION:  Generates a series of impulse data points, with the response
        of a Butterworth filter associated with each point
INPUTS: none
FUNCTIONS CALLED: Save()
CALLED BY: Butterworth()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
void Impulse()
 {
    float class,a0,a1,a2,b0,b1;
    int idum=1,i,j,bubba;

    a0=0.0676; a1=0.1352; a2=0.0676;
    b0=1.1422; b1=-0.4124;
    o[0][0]=o[1][0]=0.0;
    loopi(302)
      v[i][0] = o[i][0] = 0.;
    loopi(2)
      v[20+(i)*128][0] = 1.;
    num_vectors = 300;
    loopj(5) {
     loopi(300)
       o[i+2][0]=a0*v[i+2][0]+a1*v[i+1][0]+
              a2*v[i][0]+b0*o[i+1][0]+b1*o[i][0];
       o[0][0]=a0*v[0][0]+a1*v[299][0]+
              a2*v[298][0]+b0*o[299][0]+b1*o[298][0];
       o[1][0]=a0*v[1][0]+a1*v[0][0]+
              a2*v[299][0]+b0*o[0][0]+b1*o[299][0];
    }
    numinputs = 1;
    numoutputs = 1;
    Time_delay();
    Save();
    exit(0);
    return;
 }

/*****************************************************************************
ROUTINE NAME: Cosin()
DESCRIPTION: Generates a series of data points representing a cosine wave,
        with a Butterworth filter response associated with each data
        point.
```

```
INPUTS: none
FUNCTIONS CALLED: Save()
CALLED BY: Butterworth()
LAST UPDATED: 7 May 93                BY: Jeffrey S. Dean
*****************************************************************************/
 void Cosin()
 {

     float a0,a1,a2,b0,b1;
     int i,j;

     a0=0.0676; a1=0.1352; a2=0.0676;
     b0=1.1422; b1=-0.4124;
     o[0][0]=o[1][0]=0.0;
     loopi(128)
       v[i][0] = cos(2*3.14159*i/64);
     loopi(126)
     o[i+2][0]=a0*v[i+2][0]+a1*v[i+1][0]+a2*v[i][0]+b0*o[i+1][0]+b1*o[i][0];
     num_vectors = 128;
     numinputs = 1;
     numoutputs = 1;
     Time_delay();
     Save();
     exit(0);
     return;
 }
/*****************************************************************************
ROUTINE NAME: Step()
DESCRIPTION:  Generates a step function input, with the Butterworth filter
          response associated with each data point.
INPUTS: none
FUNCTIONS CALLED: Save()
CALLED BY: Butterworth()
LAST UPDATED: 7 May 93                BY: Jeffrey S. Dean
*****************************************************************************/
 void Step()
 {

     float class,a0,a1,a2,b0,b1;
     a0=0.0676; a1=0.1352; a2=0.0676;
     b0=1.1422; b1=-0.4124;
     loopi(150)
       o[i][0]=v[i][0]=0.0;
     loopi(30)
       v[i][0] = 0.;
```

```
    loopi(128)
      v[i+25][0] = 1.;
    loopi(126)
    o[i+2][0]=a0*v[i+2][0]+a1*v[i+1][0]+a2*v[i][0]+b0*o[i+1][0]+b1*o[i][0];
    num_vectors = 128;
    numinputs = 1;
    numoutputs = 1;
    Time_delay();
    Save();
    exit(0);
    return;
}

/*****************************************************************************
ROUTINE NAME: Clear()
DESCRIPTION: Clears the data vectors in memory, and reinitializes the vector
        count, number of inputs and outputs to zero.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
void Clear()
{

 loopi(num_vectors) {
   loopj(numinputs)
     v[i][j] = 0.;
   loopj(numoutputs)
     o[i][j] = 0.;
 }
 num_vectors = numvectors = 0;
 numinputs = numoutputs = 0;
 td = TD = 0;
 Select();
 return;
 }

/*****************************************************************************
ROUTINE NAME: Merge()
DESCRIPTION:  Allows the user to replace the inputs or outputs in memory
        with the inputs or outputs found in a data file. The number
        of data vectors in memory must match the number of vectors in
        the data file.
INPUTS: none
```

```
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                    BY: Jeffrey S. Dean
*****************************************************************************/
 void Merge()
 {
  int choice,choice2;
  int num_inputs, num_outputs;
  float junk;

  printf("\nWhat is the name of the file? :");
  skip_line;
  scanf("%s", datafile);
  printf("\n");
  printf("Reading %s . . . \n",datafile);
  ifp=fopen(datafile,"r");
  fscanf(ifp,"%d %d %d",&num_inputs,&num_outputs,&numvectors);
  printf("This file has %d inputs, %d outputs, ",num_inputs,num_outputs);
  printf("and %d vectors.\n",numvectors);
  if(num_vectors != numvectors&&&num_vectors != 0) {
    printf("**************CAN NOT CONTINUE!!***************\n");
    printf("number of vectors not the same as current data\n");
    return;
  }
  printf("Do you want to swap in: \n");
  printf("1. File inputs\n");
  printf("2. File outputs\n");
  skip_line;
  scanf("%d", &choice);
  printf("\n");
  fskip_line(ifp);
  if(choice==1){
    printf("Do you want the inputs to be used as: \n");
    printf("1. File inputs\n");
    printf("2. File outputs\n");
    skip_line;
    scanf("%d", &choice2);
    printf("\n");
    fskip_line(ifp);
    if(choice2==1){
      numinputs = num_inputs;
      printf("loading data file \n");
      loopi(numvectors) {
        loopj(numinputs)
          fscanf(ifp,"%f",&v[i][j]);
```

```
        loopj(num_outputs)
          fscanf(ifp,"%f",&junk);
      }
    }
    if(choice2==2){
      numoutputs = num_inputs;
      printf("loading data file \n");
      loopi(numvectors) {
        loopj(numinputs)
          fscanf(ifp,"%f",&o[i][j]);
        loopj(num_outputs)
          fscanf(ifp,"%f",&junk);
      }
    }
  }
  if(choice==2){
    printf("Do you want the outputs to be used as: \n");
    printf("1. File inputs\n");
    printf("2. File outputs\n");
    skip_line;
    scanf("%d", &choice2);
    printf("\n");
    fskip_line(ifp);
    if(choice2==1){
      numinputs = num_outputs;
      printf("loading data file outputs ...\n");
      loopi(numvectors) {
        loopj(num_inputs)
          fscanf(ifp,"%f",&junk);
        loopj(numoutputs)
          fscanf(ifp,"%f",&v[i][j]);
      }
    }
    if(choice2==2){
      numoutputs = num_outputs;
      printf("loading data file outputs ...\n");
      loopi(numvectors) {
        loopj(num_inputs)
          fscanf(ifp,"%f",&junk);
        loopj(numoutputs)
          fscanf(ifp,"%f",&o[i][j]);
      }
    }
  }
  fclose(ifp);
```

```
  return;
 }

/*****************************************************************************
ROUTINE NAME: Status()
DESCRIPTION:  Displays the current number of data vectors, number of inputs
          and outputs, and the time delay of the outputs.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Status()
 {
  printf("\n\n\n STATUS OF DATA: \n");
  printf("Number of vectors: %d\n", num_vectors);
  printf("%d inputs and %d outputs, ", numinputs, numoutputs);
  printf("with a time delay of %d ticks.\n", TD);
  printf("\n\n");
  return;
 }

/*****************************************************************************
ROUTINE NAME: Out_types()
DESCRIPTION:  Prints out the categories present in the data.  Assumes
          output integer represents range of categories.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void Out_types()
 {
 int sel, types[100];
  printf("OUTPUT CATEGORIES PRESENT IN DATA\n");
  printf("Which output do you want to check? (1-%d) ",numoutputs);
  skip_line;
  scanf("%d", &sel);
  printf("\n");
  printf("How many categories are there in this output? ");
  skip_line;
  scanf("%d", &cat);
  printf("\n");
  loopi(cat)
   types[i] = 0;
```

```
  loopi(num_vectors)
   types[(int)o[i][sel-1]] = 1;
  loopi(cat)
   if(types[i]==1) printf("%d ",i);
  printf("\n\n");
  return;
 }


/*****************************************************************************
ROUTINE NAME: One_cat()
DESCRIPTION:  Selects one output or one output group as valid, all other
          data vectors are classed together.  If file has two outputs,
          (as found in voice data files for this project), routine asks
          if data should be broken into subgroups (i.e. vowels, nasals,
          etc.).  If selected, the routine asks which group is to be
          used.  The routine then checks the file "phon_transl", which
          lists all 64 phonemes.  The number of phonemes in the subgroup
          is determined, and the data outputs are expanded to provide
          a category for each phoneme in the group.  If the outputs are
          not to be broken into subgroups or there are more than 2 outputs,
          the routine assumes that the outputs represent categories, and
          prompt the user to select one output.  The routine then creates
          two output categories for the data, one for the selected
          category and one for all other data vectors.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
 void One_cat()
 {
  int index, junk, out_vect[64], cnt, sub, cont;
  float cat_data;
  char choice, out_vect_s[64][5], group[10], phon[10], sel_group[10];

  choice = NULL;
  if(numoutputs==2) {
    printf("Do you need it broken into subgroups? (y/n) ");
    skip_line;
    scanf("%c", &choice);
    printf("\n");
    if(choice=='y') {
      printf("Which subgroup do you want? (0 - 5)\n");
      skip_line;
```

```
    scanf("%d", &sub);
    printf("\n");
    ifp = fopen("phon_transl", "r");
    count = 0;
    loopi(63) {
        fscanf(ifp,"%d %s %d %s",&junk,phon,&cat,group);
        if(cat==sub) {
           strcpy(sel_group, group);
           out_vect[count] = junk;
           strcpy(out_vect_s[count],phon);
           count++;
        }
    }
    loopi(num_vectors) {
       cnt = 0;
       cat_data = o[i][0];
       loopj(count) {
           if((int)cat_data==out_vect[j]) {
             o[i][j] = 1.;
             cnt++;
           }
           else o[i][j] = 0.;
       }
       if(cnt==0) o[i][count] = 1.;
    }
    numoutputs = count+1;
    printf("The selected category is %s\n",sel_group);
    loopi(count)
      printf("%d ",out_vect[i]);
    printf("\n");
    loopi(count)
      printf("%s ",out_vect_s[i]);
    printf("\n");
  }
}
else {
   printf("Which category do you want? (1 - %d)\n",numoutputs);
   skip_line;
   scanf("%d", &sel);
   printf("\n");
   loopi(num_vectors) {
      if(o[i][sel-1]==1) {
        o[i][0] = 1.;
      /* o[i][1] = 0.; */
      }
```

```
        else {
          o[i][0] = 0.;
        /* o[i][1] = 1.; */
        }
      }
      numoutputs = 1;
    }
    printf("Press <return> to continue\n");
    skip_line;
    scanf("%c", &cont);
    return;
  }
/*****************************************************************************
ROUTINE NAME: Phoneme()
DESCRIPTION: Checks data to determine which phonemes are present
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                    BY: Jeffrey S. Dean
*****************************************************************************/
void Phoneme()
{
int phon[6][64];
char cont;

      loopi(6)
        loopj(64)
            phon[i][j] = 0;

      loopi(num_vectors)
        phon[(int)o[i][1]][(int)o[i][0]] = 1;

      printf("Members of the nasal phoneme group are:\n");
      loopi(64)
        if(phon[0][i]==1) printf("%d  ",i);
      printf("\n");

      printf("Members of the vowel phoneme group are:\n");
      loopi(64)
        if(phon[1][i]==1) printf("%d  ",i);
      printf("\n");

      printf("Members of the stop phoneme group are:\n");
      loopi(64)
        if(phon[2][i]==1) printf("%d  ",i);
```

```
    printf("\n");

    printf("Members of the fricative phoneme group are:\n");
    loopi(64)
      if(phon[3][i]==1) printf("%d  ",i);
    printf("\n");

    printf("Members of the silence phoneme group are:\n");
    loopi(64)
      if(phon[4][i]==1) printf("%d  ",i);
    printf("\n");

    printf("Members of the liq-glide phoneme group are:\n");
    loopi(64)
      if(phon[5][i]==1) printf("%d  ",i);
    printf("\n");
    printf("Press <return> to continue\n");
    skip_line;
    scanf("%c", &cont);
    return;
}
/*****************************************************************************
ROUTINE NAME: Compare()
DESCRIPTION: Determines which input value is the largest, then check to see
        whether corresponding output value is a 1.  Used to compare net
        outputs with the desired outputs; checks net accuracy for each
        output category.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
void Compare()
{
    float max;
    int high_out;
    int score[64][2];
    char cont;

    loopi(64)
      loopj(2)
        score[i][j] = 0;

    if(numinputs!=numoutputs) {
      printf("Different number of inputs and outputs. Can't compare.\n");
```

```
    return;
  }
  loopi(num_vectors) {
    max = -1000.;
    loopj(numinputs)
     if(v[i][j]>max) {
       max = v[i][j];
       high_out = j;
     }
    loopj(numinputs)
       if(o[i][j] == 1) score[j][0]++;
    if(o[i][high_out] == 1)
        score[high_out][1]++;
  }
  loopi(numinputs) {
    printf("Category %d: %d examples",i,score[i][0]);
    if(score[i][0]>0)
      printf(", %f%% correct\n",((float)score[i][1]/score[i][0])*100);
    printf("\n");
  }
  printf("Press <return> to continue\n");
  skip_line;
  scanf("%c", &cont);
  return;
}

/*****************************************************************************
ROUTINE NAME: Differentiate()
DESCRIPTION: Replaces inputs in each data vector with the difference between
        the inputs, then replaces inputs in each data vector with the
        difference between the input and the next data vector input.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: File_work()
LAST UPDATED: 7 May 93                  BY: Jeffrey S. Dean
*****************************************************************************/
void Differentiate()
{
  loopi(numvectors) {
    loopj(numinputs-1)
       v[i][j]= v[i][j+1] - v[i][j];
    v[i][0] = 0.;
  }

  loopi(numvectors-1)
```

```
        loopj(numinputs)
            v[i][j] = v[i+1][j] - v[i][j];

      loopj(numinputs)
        v[0][j] = 0.;
      return;
}

/*****************************************************************************
ROUTINE NAME: Xor()
DESCRIPTION: Creates XOR training/test data for the neural net, with either integer or
floating point values.
INPUTS: none
FUNCTIONS CALLED: none
CALLED BY: Select()
LAST UPDATED: 7 May 93                    BY: Jeffrey S. Dean
*****************************************************************************/
void Xor()
{
      float class, seed;
      int idum=1, choice;

   printf("\nEnter random number seed\n");
   scanf("%d", &seed);
   idum = -IABS(seed);
   printf("\nDo you wish:\n1. Integer values\n2. Floating point values\n");
   scanf("%d", &choice);
   printf("\n");
   loopi(1024) {
    loopj(2) {
      v[i][j]=ran1(&idum);
      if(choice==1)  {
          if (v[i][j]>0.5) v[i][j]=1.0;
          else v[i][j]=0.0;
      }
    }

          if ((v[i][0]>0.5) && (v[i][1]>0.5))
            o[i][0]=0.0;
          if ((v[i][0]<=0.5) && (v[i][1]>0.5))
            o[i][0]=1.0;
          if ((v[i][0]>0.5) && (v[i][1]<=0.5))
            o[i][0]=1.0;
          if ((v[i][0]<=0.5) && (v[i][1]<=0.5))
            o[i][0]=0.0;
```

```
    }
    numoutputs = 1;
    num_vectors = 1024;
    numinputs = 2;
    Time_delay();
    Save();
    exit(0);
    return;
}
/*****************************************************************************
ROUTINE NAME: Sequence()
DESCRIPTION: Allows the user to select function for dealing with sequentially related
data.  Functions include:
1. Converting codeword sequences (seq length   codeword1   codeword2 ...) into net
format
2. Randomize training/test sequences, so that sequence categories are mixed
3. Convert sequences with Fourier coefficient inputs to net format
4. Read in "sequence.dat" file and check accuracy of net output
INPUTS: none
FUNCTIONS CALLED: Convert(), Randomize(), Fourier_input(), and Score_seq()
CALLED BY: Select()
LAST UPDATED: 7 Jan 94                  BY: Jeffrey S. Dean
*****************************************************************************/

void Sequence()
{

 int choice;
 num_vectors = numvectors = 0.;
 for (;;) {
   printf("Choose one of the following: \n");
   printf("\n1. Convert codeword sequences to net format \n");
   printf("\n2. Randomize sequence of codeword strings \n");
   printf("\n3. Convert Fourier magnitude sequences to net format\n");
   printf("\n4. Score the accuracy of a sequence.dat file\n");
   scanf("%d", &choice);
   printf("\n");

   if(choice==1)  Convert();
   if(choice==2)  Randomize();
   if(choice==3)  Fourier_input();
   if(choice==4)  Score_seq();
 }
 return;
}
```

```
/*****************************************************************************
ROUTINE NAME: Convert()
DESCRIPTION: Convert codeword sequences to net format
INPUTS: none
FUNCTIONS CALLED: File_work()
CALLED BY: Sequence()
LAST UPDATED: 7 Jan 94                BY: Jeffrey S. Dean
*****************************************************************************/

void Convert()
{
   int t, categ, length, sect;
   float junk;
   TD = 0;
   printf("\nWhat is the name of the file? :");
   skip_line;
   scanf("%s", datafile);
   printf("\n");
   printf("\nWhich category does this belong to? :");
   skip_line;
   scanf("%d", &categ);
   printf("\n");
    ifp=fopen(datafile,"r");
    t= 0;
    loopk(4)
     loopi(50) {
          fscanf(ifp, "%d", &length);
          loopj(length) {
             fscanf(ifp,"%f", &v[t][0]);
             loopl(6)
                 o[t][l] = 0;
             o[t][categ] = 1;
             t++;
          }
          loopj(6) {
             v[t][0] = -1;
             loopl(6)
                 o[t][l] = 0;
             o[t][0] = 1;
             t++;
          }
     }
   num_vectors = t - 1;
   numinputs = 1;
   numoutputs = 6;
```

```
    fclose(ifp);
    File_work();
    num_vectors = 0;
    return;
}
/*****************************************************************************
ROUTINE NAME: Randomize()
DESCRIPTION: Randomize sequences for training/test data
INPUTS: none
FUNCTIONS CALLED: File_work()
CALLED BY: Sequence()
LAST UPDATED: 7 Jan 94                BY: Jeffrey S. Dean
*****************************************************************************/

void Randomize()
{

   int choice, idum=15756, junk, range, incr, max;
   printf("\nWhat is the name of the file? :");
   skip_line;
   scanf("%s", datafile);
   printf("\n");
   printf("Reading %s . . . \n",datafile);
   ifp=fopen(datafile,"r");
   fscanf(ifp,"%d %d %d %d
%d",&numinputs,&numoutputs,&junk,&numvectors,&TD);
   num_vectors = numvectors;
   loopi(1000)
      pick[i][0] = 0;
   loopi(numvectors) { /* Load in data */
     loopj(numinputs)
       fscanf(ifp,"%f",&v2[i][j]);
     loopj(numoutputs)
       fscanf(ifp,"%f",&o2[i][j]);
   }
   incr = 0;
   pntr[0][0] = 0;
   count = 1;
   printf("Examining sequence starting positions\n");
   loopi(numvectors-1) /* find out where sequences start */
     if(o2[i+1][0] != 1. && o2[i][0] == 1.) {
          pntr[count][0] = i+1;
          /* printf("%d  %d\n ",pntr[count][0], count); */
          count++;
     }
```

```
  max = count;
  pntr[max][0] = numvectors;
  printf("\n\n");
  count = 0;
  printf("Randomizing sequences\n");
  loopi(max) { /* Pick one of sequences in file randomly */
    for(;;) {
      choice = (int)(max*ran1(&idum));
      if(choice>=0&&choice<max)
        if(pick[choice][0]==0.)
          break;
    }
    pick[choice][0] = 1.; /* Identify sequence as picked */
    range = pntr[choice+1][0]-pntr[choice][0];
    /*
    printf("%d %d %d %d\n", incr,choice, pntr[choice][0], range);
    */
    incr++;

    /* Loop from beginning of this sequence to next */
    loopj(range) {
     loopk(numinputs)
        v[count][k] = v2[pntr[choice][0]+j][k];
     loopk(numoutputs)
        o[count][k] = o2[pntr[choice][0]+j][k];
     count++;
    }
  }
  fclose(ifp);
  File_work();
  return;
}
/*****************************************************************************
ROUTINE NAME: Fourier_input()
DESCRIPTION: Reads in sequences of 28 Fourier amplitude coefficients and converts
them to network input format.
INPUTS: none
FUNCTIONS CALLED: File_work()
CALLED BY: Sequence()
LAST UPDATED: 7 Jan 94                BY: Jeffrey S. Dean
*****************************************************************************/

 void Fourier_input()
 {
```

```
    int Start, categ, t;
    printf("\nWhat is the name of the file? :");
    skip_line;
    scanf("%s", datafile);
    printf("\n");
    printf("\nWhich category does this belong to? :");
    skip_line;
    scanf("%d", &categ);
    printf("\n");
    printf("Reading %s . . . \n",datafile);
    ifp=fopen(datafile,"r");

      Start = 14;
      t = 0;
      loopi(4) {
        loopj(50) {
          loopk(Start) {
            loopl(28)
              fscanf(ifp,"%f",&v[t][l]);
            loopl(4)
              o[t][l] = 0.;
            o[t][categ] = 1.;
            t++;
          }
          loopk(4) {
            loopl(28)
              v[t][l] = 0.;
            loopl(4)
              o[t][l] = 0.;
            o[t][0] = 1.;
            t++;
          }
        }
        Start += 2;
      }
  num_vectors = t;
  numinputs = 28;
  numoutputs = 4;
  fclose(ifp);
  File_work();
  return;
}
/*****************************************************************************
ROUTINE NAME: Score_seq()
```

```
DESCRIPTION: Scores network on accuracy in determining sequence category, based on
netwwork response on last ten points each sequence
INPUTS: none
FUNCTIONS CALLED: File_work()
CALLED BY: Sequence()
LAST UPDATED: 7 Jan 94                BY: Jeffrey S. Dean
**************************************************************************/

void Score_seq()
{
  int check[10], index, count, max, total[10], good[10], num_sequence;
  int num_seq;
  float score;
  ifp=fopen("sequence.dat","r");
  fscanf(ifp,"%d %d %d",&num_inputs,&num_outputs,&num_vectors);
  printf("This file has %d inputs, %d outputs, ",num_inputs,num_outputs);
  printf("and %d vectors.\n",num_vectors);
  fskip_line(ifp);
  printf("loading data file ...\n");
  loopi(num_vectors) {
        fscanf(ifp,"%f",&v[i][0]);
        fscanf(ifp,"%f",&o[i][0]);
          if(v[i][0]>9||v[i][0]<0) printf("Out of bounds, line %d\n",i);
          if(o[i][0]>9||o[i][0]<0) printf("Out of bounds, line %d\n",i);
  }
  fclose(ifp);
  printf("\nData file loaded. \n");
  num_sequence = 0;
  loopi(num_vectors) {
      if(o[i][0] == 0&&o[i+1][0]!=0)
       num_sequence++;
  }
  printf("There are %d sequences\n",num_sequence);
  loopi(10) {
      good[i] = 0;
      total[i] = 0;
  }
  num_seq = 0;
  printf("Starting to process sequences\n");
  i = 0;
  loopk(num_sequence) {
     while(o[i][0] == 0.)    /* Move to next sequence */
             i++;
     loopj(10)        /* Zero count of categories for sequence */
        check[j] = 0;
```

```
        count = 0;
        while(o[i][0] != 0&&i<num_vectors) {   /* While in a sequence */
                if(count>10)
            check[(int)v[i][0]] += 1; /* Tally outputs of net */
           i++;          /* Increment to next position */
           count++;     /* Count length of sequence */
        }

          num_seq++;
        max = -1;
        loopj(10)         /* find out which output most often */
           if(check[j]>max) { /* chosen by net for this sequence */
             max = check[j];
             index = j; /* Index is most chosen category */
           }

        if(index == o[i-1][0]) /* If index is right answer */
           good[index]++;      /* Show that category was scored */
        total[(int)o[i-1][0]]++; /* correctly.  Inc. count of category */

    }

    count = 0;
    loopi(10)
        count += good[i];
    score = 100*(float) count/(float) num_sequence;
    printf("The net scored %f%% of the sequences correctly\n\n", score);
    loopi(10)
        if(total[i] > 0) {
            score = 100*(float) good[i]/(float) total[i];
            printf(" Category %d was scored %f%% correctly\n", i, score);
        }
    exit(0);
    return;
}
#undef IC3
#undef M1
#undef IA1
#undef IC1
#undef RM1
#undef M2
#undef IA2
#undef IC2
#undef RM2
```

```
#undef M3
#undef IA3

/* def.h ************************************************************

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

     date: 11 Jun 93

     written by: Jeffrey S. Dean
*********************************************************************************/
int *vector();
float **matrix();
int **imatrix();

FILE *ifp, *ofp;
char str[80], *datafile[20], *outfile[20];
int i, j, k, cat, sel;
int numinputs, numoutputs, numnodes, numvectors, num_vectors, td, TD;
int num_inputs, num_outputs,count;
int *pick;
float **v, **o;
void Butterworth();
void File_work();
void Append();
void Time_delay();
void Categories();
void Cull();
void Norm();
void View();
void Random();
void Cosin();
void Step();
void Save();
void Impulse();
void Clear();
void Merge();
void Status();
void Out_types();
void One_cat();
void Phoneme();
void Compare();
void Differentiate();
void Xor();
```

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


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

FFT.C - Fast Fourier Transform program

*************************************************************/
#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(argc,argv)
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 forma
t */
      /*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
```

Appendix D. *Payton Auditory Model*

One of the functions that is always cited as a potential use for recurrent neural networks is speech analysis. Because of the grammatical rules inherent in language, speech naturally has a sequential structure that can be learned by a recurrent network, which can learn temporal probabilities as well as the spatial (frequency) probabilities calculated by a standard backprop net. The speech data used to train the net can be generated in several ways. One standard method is to Fast Fourier Transform (FFT) the sampled and digitized speech, and use the Fourier coefficients as the training data for the network. Variations on this approach include using Cepstral, Discrete Cosine Transform (DCT) or wavelet coefficients. All of these approaches are based on transform algorithms that convert the temporal domain information into a frequency domain. Each of these approaches have their advantages and disadvantages.

In the same way that neural network designs are inspired by how neurons work in living systems, many researchers have been trying to emulate the way in which the hearing systems in animals process sound energy into information encoded in the auditory nerves. The acoustical mechanics of the ear allow us to pick out one voice among many, to make sense out of the series of vowels and consonants we hear with relatively high reliability. The ear works in a very non-linear way to pick out the formants, or peak frequency points, which are critical in understanding speech.

The Payton(8) auditory model is one of many algorithms(5)(8) that model the way in which our auditory systems convert sound into neural impulses. This model produces 20 outputs, that correspond to the predicted activity of 20 cochlear neurons that carry sound information to the brain in a living mammal.

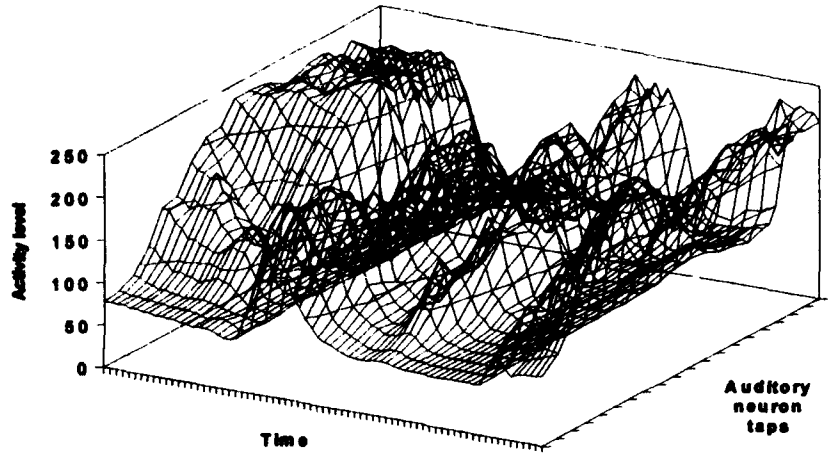


Figure 31: A plot of voice data preprocessed by the Payton algorithm. Note the peaks in the data representing the speech formants.

Bibliography

1. Allred, Lloyd G. and Gary Kelly, "Supervised Learning Techniques for Backpropagation Networks'" *Proceedings of the International Joint Conference on Neural Networks, I,* pages 721-728, 1990.

2. Ellman, Jeffrey L., "Finding Structure in Time," *Cognitive Science*, University of California, San Diego, pages 179-211, 1990

3. Fielding, K. H. and others, "Spatio-temporal Pattern Recognition Using Hidden Markov Models", *Proceedings of the SPIE Vol. 2032 Neural and Stochastic Methods in Image and Signal Processing II*, pages 144-154, July 1993.

4. Fang, Yan and Terrence Sejnowski. "Faster Learning for Dynamic Recurrent Backpropagation," *Neural Computation, 2,* pages 270-273 (1990)

5. Hewitt, Michael J. and Ray Meddis, "An Evaluation of Eight Computer Models of Mammalian Inner Hair-cell Function," *Journal of the Acoustical Society of America,* pages 904-917, August 1991.

6. Jordan, M. I., "Serial Order: A Parallel Distributed Processing Approach," *Institute For Cognitive Sciences Report 8604*, University of California, San Diego, 1986

7. Lindsey, Randall L, *Function Prediction Using Recurrent Neural Networks,* MS Thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

8. Payton, Karen L. "Vowel Processing by a Model of the Auditory Periphery: A Comparison to Eighth-Nerve Responses," *Journal of the Acoustical Society of America,* pages 145-162, January 1988.

9. Pearlmutter, B. A. "Learning State Space Trajectories in Recurrent Neural Networks," *Proceedings of the International Joint Conference on Neural Networks,* Vol 2, pages 365-372, 1989.

10. Pineda, F. J. "Generalization of Back-Propagation to Recurrent Neural Networks," *Physical Review Letters,* 59-19, pages 2229-2232, November 1987

11. Press, William H. and others, *Numerical Recipes in C.* Cambridge: The MIT Press, 1991.

12. Rogers, Steven K. and Matthew Kabrisky. *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition.* Washington, SPIE Optical Engineering Press, 1991.

13. Rohwer, R. and Bruce Forrest. " Training Time Dependence in Neural Networks," *Proceeding of the IEEE First international Conference on Neural Networks, II,* pages 701-708, June 1987

14. Rumelhart, David E. et al. "Learning Internal Representations by Error Propagation," *Parallel Distributed Processing: Explorations in the Microstructure of Cognition,* Volume VI. Cambridge: The MIT Press, pages 318-362, 1986

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

16. Van Ooyen, A. and B. Nienhuis, "Improving the Convergence of the Back-Propagation Algorithm," *Neural Networks*, Vol 5, pages 465-471, 1992

17. Williams, R. J. and David Zipser, "A Learning Algorithm For Continually Running Fully Recurrent Neural Networks," *Neural Computation, 1,* pages 270-280, 1989.

18. Waibel, Alexander and others. "Parallelism, Hierarchy, Scaling in Time-Delay Neural Networks for Spotting Japanese Phonemes/CV-Syllables,",*Proceedings of the International Joint Conference on Neural Networks, II,* pages 81-88, 1989.

19. Waibel, Alexander and others. "Phoneme Recognition Using Time-Delay Neural Networks," *IEEE Transactions on Acoustics, Speech and Signal Processing,37-3,* pages 328-339, March 1989.

20. Zipser, David. "A Subgrouping Strategy That Reduces Complexity and Speeds Up Learning in Recurrent Networks," *Neural Computation, 1*, pages 552-558, 1989.

Vita

Jeffrey S. Dean was born on 24 September 1956 in Hackensack, N.J. In 1974 he graduated from Pine Bush High School and matriculated to Saint Louis University, in Saint Louis, MO. He graduated from SLU in 1979 with a B.A. in Biology and Chemistry, and taught these subjects in a Catholic high school for two years. Mr. Dean enlisted in the Air Force in 1981 under the Engineering Conversion program through AFIT. After OTS training he was assigned to the University of Missouri at Columbus, MO for two years to obtain an Electrical Engineering B.S. in December 1983. After serving five years as an electrical engineer at the World Wide Airborne Command Post System Program Office at Tinker AFB, he was assigned to Wright-Patterson AFB in the Aeronautical Systems Command Flight Systems Engineering Directorate (ASD/ENF). While there Capt Dean served as group leader for a Directorate computer support group, and later as a chief functional engineer in the Productivity Reliability Availability Maintainability (PRAM)/ Reliability and Maintainability Technology Insertion Program (RAMTIP) office. During the four years assigned at Wright-Patterson AFB, he took AFIT classes as a part time student. He separated from the Air Force in November 1992, with all class work completed except for the thesis, which was finished while working as a civilian at Kelly AFB in the Advanced Diagnostics Technology Insertion Center (ADTIC).

Mr. Dean currently lives in San Antonio, Texas with his wife Marla and his five children (Heather, Hillary, Thomas, Kayla and Meghan), in a house big enough to hold them all.

Permanent address: 6021 Cammie Way
San Antonio, TX 78238

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1994	Master's Thesis

4. TITLE AND SUBTITLE
Subgrouped Real Time Recurrent Learning Neural Networks

5. FUNDING NUMBERS

6. AUTHOR(S)
Jeffrey S. Dean, Capt USAFR

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

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/ENG/GE/94J-01

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Diagnostics Technology Insertion Center (ADTIC)
SA-ALC/LDAE
Kelly AFB, TX 78241

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 subgrouped Real Time Recurrent Learning (RTRL) network was evaluated. The one layer net successfully learns the XOR problem, and can be trained to perform time dependent functions. The net was tested as a predictor on the behavior of a signal, based on past behavior. While the net was not able to predict the signal's future behavior, it tracked the signal closely. The net was also tested as a classifier for time varying phenomena; for the differentiation of five classes of vehicle images based on features extracted from the visual information. The net achieved a 99.2% accuracy in recognizing the five vehicle classes.

The behavior of the subgrouped RTRL net was compared to the RTRL network described in Capt R. Lindsey's AFIT Master's thesis. The subgrouped RTRL performance proved close to the RTRL network in accuracy while reducing the time required to train the network for multiple output (classification) problems.

14. SUBJECT TERMS
neural network, recurrent, RTRL, image recognition
time dependence, temporal, sequence recognition

15. NUMBER OF PAGES
154

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 Names(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/Availablity Statement. Denote public availability or limitation. 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 - DOD - Leave blank
DOE - DOE - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports
NASA - NASA - Leave blank
NTIS - 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.