Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

ABSTRACT

Classification of Patterns in EEG Recordings: A Comparison of Back-propagation Networks vs. Predictive Autoencoder Networks

by
Brian Armieri

Recent research exploring the use of neural networks for electro-encephalogram (EEG) pattern classification has found that a three-layer back-propagation network could be successfully trained to identify high voltage spike-and-wave spindle (HVS) patterns caused by epileptic seizures (Jando *et. al.*, in press). However, there is no reason to predict that back-propagation is the best possible network architecture for EEG classification. A back-propagation neural network and a predictive autoencoder neural network were compared to determine which network was better at correct classifying both HVS and non-HVS patterns.

Both networks were able to classify 88%-89% of all patterns using a limited set of training data. The predictive autoencoder network trained with less epochs and appeared more resistant to overtraining. However, performance of the predictive autoencoder network may vary if it is stopped before it has trained for a sufficient number of epochs.

CLASSIFICATION OF PATTERNS IN EEG RECORDINGS: A COMPARISON OF BACK-PROPAGATION NETWORKS VS. PREDICTIVE AUTOENCODER NETWORKS

by
Brian Armieri

Submitted to the Faculty of the
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

May 1993

APPROVAL PAGE

Classification of Patterns
in EEG Recordings:
A Comparison of
Back-propagation Networks
vs.
Predictive Autoencoder
Networks

Brian Armieri

May 1993

Dr. Mark Gluck, Thesis Adviser
Assistant Professor, Center for Molecular and Behavioral Neuroscience, Rutgers University

---

Dr. Catherine Myers, Center for Molecular and Behavioral Neuroscience, Rutgers University

---

Dr. Peter Ng, Chairperson, Department of Computer and Information Science, New Jersey Institute of Technology

BIOGRAPHICAL SKETCH

Author: Brian Armieri

Degree: Master of Science in Computer Science

Date: May, 1993

Undergraduate and Graduate Education:

- Master of Science in Computer Science,
  New Jersey Institute of Technology, Newark, NJ, 1993

- Bachelor of Arts in Psychology,
  Cornell University, Ithaca, NY, 1990

Major: Computer Science

This thesis is dedicated in memory of Patrick Armieri

ACKNOWLEDGMENT

The author would like to express his sincere gratitude to Dr. Mark Gluck and Dr. Catherine Myers for their continuous guidance and friendship throughout this research.

Dr. Gabor Jando also deserves appreciation for many tireless explanations of his research and for his help with the EEG data files.

Finally, thanks go to Dr. Jim McHugh for suggesting research opportunities at the Center for Molecular and Biological Neuroscience.

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

CHAPTER 1. INTRODUCTION

1.1 Overview

Recent research has explored the use of neural nets for electroencephalogram (EEG) pattern classification (Jando *et. al.* , 1993). Results indicated that a three-layer back-propagation network could be successfully trained to identify high voltage spike-and-wave spindle (HVS) patterns caused by epileptic seizures.

To date, the back-propagation network has been used to classify EEG patterns because its mathematical background is well understood and because there is a large amount of information regarding its efficiency in the domain of pattern recognition tasks. However, there is no reason to predict that back-propagation is the best possible network architecture for this particular problem.

Gluck and Myers (1992, 1993) have described how an extension to the encoder network architecture (Hinton, 1989), called a predictive autoencoder, can be used to develop a network which models the function of the hippocampus in human cognition. The predictive autoencoder is also potentially applicable to the problem of computer-based detection of epileptic seizures.

This research compares the performance of the back-propagation neural network against the performance of the predictive autoencoder in recognizing HVS patterns in EEG data. The design and testing of simulations for the two network architectures is described, and the results are analyzed to examine the performance of the predictive autoencoder on a real-world classification task.

1.2 Neural Networks

1.2.1 Nodes

While computers can outperform the human brain in areas such as mathematical computation, the brains of even the smallest animals are better than computers at complex pattern analysis problems like visual recognition. Neural network algorithms attempt to simulate the physiology of a brain in order to improve the computer's performance on these type of tasks.

A neural network is a collection of processing elements, usually called nodes, which accept many inputs, compute a weighted summation of the input values, and finally generate a single output value which can fan out to many other processing elements in the network. Figure 1.1 displays the details of a typical processing element.

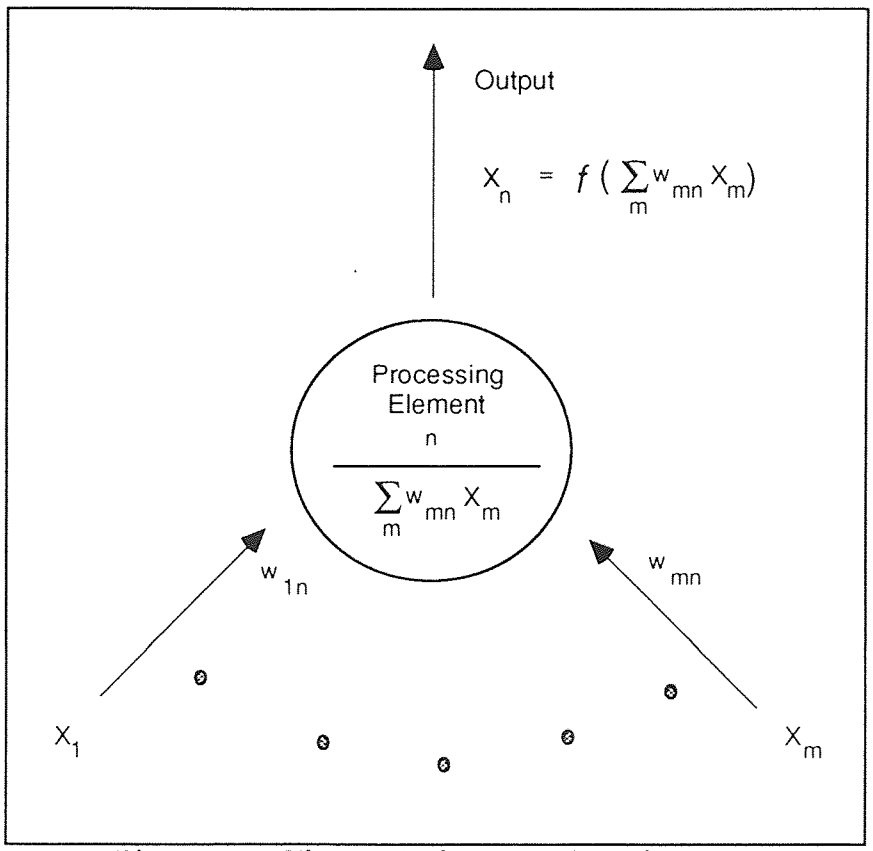

Figure 1.1 The typical processing element.

Each input to the processing element is associated with a weight, which corresponds to the strength of the synaptic connection in a biological system. The weight from node m to node n is denoted by w_{mn}. In most networks, the weight can take on a positive or negative value.

In figure 1.1, the output value for a processing element is determined by feeding the net weighted input value to an output function. Different output functions have been proposed for different neural network architectures (Caudill & Butler, 1990). The three most common types of functions are linear, threshold, and sigmoid:

> For linear functions, the output activity is proportional to the total weighted input. For threshold units, output is set at one of two levels, depending on whether the total input is greater than or less than some value. For sigmoid units, the output varies continuously but not linearly as the input changes. Sigmoid units bear a greater resemblance to real neurons than do linear or threshold units, but all three must be considered rough approximations (Hinton, 1992, p. 145).

A processing element can learn to classify patterns by adjusting input weights w_{mn} according to some learning law; in this research, the backpropagation and predictive autoencoder networks use the generalized delta rule (Rumelhart & McClelland, 1986). The generalized delta rule is discussed in more detail in section 1.2.4.

1.2.2 Feed-Forward Networks

Neural networks further simulate brain physiology by combining individual processing elements into interconnected layers. A hierarchical layer structure of two, three, or more nodes can solve more complex analysis

problems than a single processing element. If every node in layer n is connected to every node in layer n+1, then the network is said to be fully connected.

An example of a fully-connected three-layer network of processing elements is shown in figure 1.2. Processing elements in a layer without direct connection to either the input or output of the network, such as the middle layer of nodes in figure 1.2, are called hidden units.

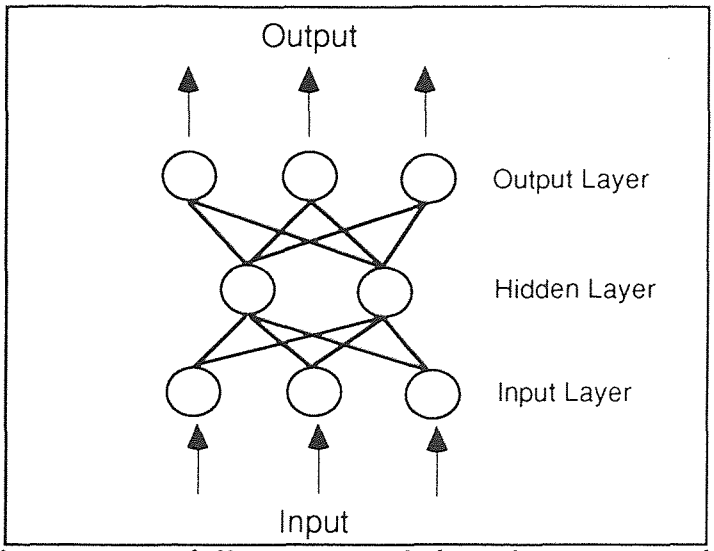

Figure 1.2 A fully-connected three-layer network.

A network is said to be a feed-forward network if the activation of nodes proceeds in a single direction from the input layers nodes to the output layer nodes. Each node in a layer calculates a weighted summation of its inputs:

$$s_n = \sum_{m=1}^{I} (w_{mn} \cdot X_m) \tag{1.1}$$

where s_n is the summation of the weighted input values for node n, I is the number of inputs to the node, w_{mn} is the weight from input m to the node n, and X_m is the input value m.

If a sigmoid output function is used, the final output value for a node is calculated using the results from equation 1.1 as the parameter in the sigmoid function:

(1.2) $$X_n = (1 + e^{-s_n})^{-1}$$

where X_n is the output value for node n.

Equation 1.2 is used to calculate an output value for every node in a layer. Those values are then the input values to the nodes in the next layer. This process continues until outputs are determined from all output layer nodes.

1.2.3 Training

Neural networks are usually first put through a training phase, where sample inputs are presented to the network. During training, the weights (which are initially set to random values) are adjusted according to the learning law. If the network is supplied with the desired output for each training example, the network is said to use supervised learning. Both of the networks used in this research are examples of supervised learning algorithms. However, there are also general-purpose unsupervised learning algorithms (Freeman & Skapura, 1991; Hinton 1989) which learn large sets of patterns without being given information about how to classify them. Neural networks using unsupervised learning algorithms intuitively seem to simulate biological systems more closely.

1.2.4 Backpropagation

Backpropagation networks are usually implemented as three-layer, fully interconnected networks. In most cases, the input nodes simply feed the input values to the hidden layer. Backpropagation networks "learn" by

adjusting weights to the hidden and output layers. Since back-propagation is a supervised learning process, the training phase consists of presenting example patterns together with corresponding desired output patterns.

For each example pattern, the feed forward process determines the network output using the current weights. Then, using the generalized delta rule (Rumelhart & McClelland, 1986), the network weights are adjusted:

Step 1. For each output node n, calculate the error β_n:

(1.3) $$\beta_n = (y_n - X_n)$$

where y_n is the desired output for output node n.

Step 2. For all other nodes (from hidden layer to input layer) calculate the error β_m using the calculation from the layer after it:

(1.4) $$\beta_m = \sum_{n=1}^{N} (w_{mn} \cdot X_n \cdot (1 - X_n) \cdot \beta_n)$$

where N is the number of nodes in the next layer.

Step 3. For every weight in the network, adjust the weight according to the formula:

(1.5) $$w_{mn} = w_{mn} + \lambda \cdot \beta_m$$

where λ is the learning coefficient (a number between 0 and 1).

Ideally, steps 1 through 3 are repeated for every example pattern in the training set until the network is trained. At that point, it is often desirable to check performance of the network on a new, but similar, set of data (the transfer set). This helps to give some measure of how well the network performance will generalize to non-training data.

The backpropagation algorithm has been well studied, and the issues involved in implementing backpropagation nets (e.g., improving algorithm performance, setting the learning coefficient, choosing representative training sets) have been well documented. The reader is referred to Freeman and Skapura (1991) for an overview of these topics.

1.2.5 Predictive Autoencoder Networks

Hinton (1989) describes how "self-supervised backpropagation" can be achieved using a multi-layer encoding network. The backpropagation algorithm is still used to train an encoder network. However, during the training phase the desired output patterns are identical to the input pattern.

Usually, the encoder has the same number of input and output nodes. If the middle layer contains fewer nodes than the input (and output) layers, the network is forced to "compress" features of the input pattern.

Gluck and Myers (1992, 1993) used a variation of the encoder network to model the role of the hippocampus in human learning. The predictive autoencoder (figure 1.3) adds an extra output node to the encoder that can be trained for classification and prediction.

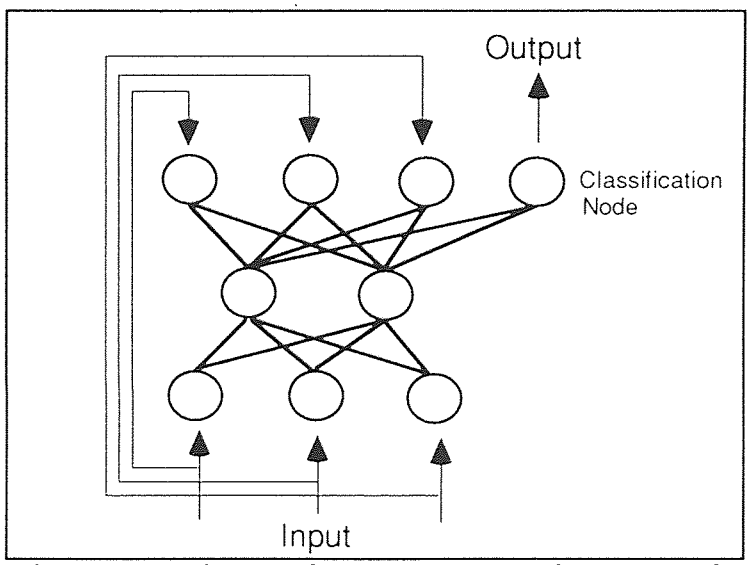

Figure 1.3 The predictive autoencoder network.

While the standard backpropagation network can settle on an arbitrary set of weights to produce a desired classification, the predictive autoencoder is constrained to use weights which are also useful in reconstructing the features of its input patterns.

1.3 EEG Classification with Neural Networks

Jando *et. al.* (in press) used a backpropagation network to classify HVS (high voltage spike and wave) patterns caused by epileptic seizures. HVS patterns were chosen because they are easy to recognize visually and because a large database of EEG readings from epileptic rats had been accumulated. An HVS period is depicted in figure 1.4.

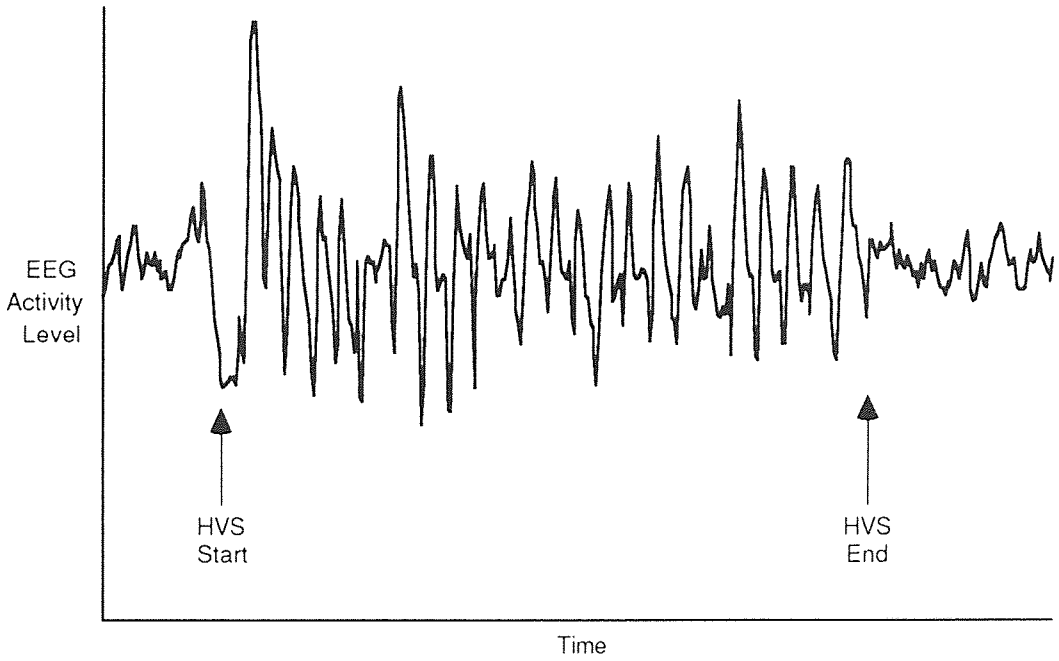

Figure 1.4 An example of an HVS event in EEG output.

Jando *et. al.* varied the number of input, hidden, and output nodes to determine the optimal network structure. The most effective network was found to have 16 input nodes, 19 hidden nodes, and one output node.

In order to classify patterns into HVS or non-HVS categories, a threshold was applied to the output node to determine the occurrence of an HVS event. If the output node's activation was above the threshold, the input pattern was classified as an HVS event.

After training, HVS events were successfully detected at a rate of 93% - 99%, depending on the threshold level. Falsely detected events (false positives) varied between 18%-40% of non-HVS events.

Jando *et. al.* reported that backpropagation was chosen largely due to the fact that backpropagation has been widely applied to the problem of pattern recognition. However, it was mentioned that other types of neural networks might show even better performance than standard backpropagation.

CHAPTER 2. MATERIALS AND METHODS

2.1 Data Collection & Preprocessing

The same database of EEG recordings used by Jando *et. al.* (1993) was used in this study. The entire database consists of 12 hour recordings of EEG activity from two hundred and fifty rats specifically bred for epilepsy. For a detailed description of the animals, surgery, and recording procedures, see Jando *et. al.* (1993). For the purposes of this thesis paper, it suffices to say that Jando *et. al.* selected right frontal cortical electrode data from 16 rats (representative of the larger sample) and manually detected HVS events in each of the 16 data files using a mouse-guided editor.

The purpose of the research presented here was to compare the back-propagation and predictive autoencoder networks. Therefore, it was sufficient to randomly select one of the 16 manually-marked data files for analysis. The data file was further broken down into a randomly-selected set of 512 training patterns (approximately 50% HVS patterns and 50% non-HVS patterns) and a set of 1000 transfer patterns.

Since Jando *et. al.* had reported that back-propagation results were more successful when using fast Fourier transformed (FFT) data, FFT data files were created for both the training set and the transfer set. Both raw data files and FFT data files were analyzed in this study.

All of the data files (raw vs. FFT, training vs. transfer) were normalized using a simple linear normalization function:

$$f(x) = (x - \alpha_{min}) \div (\alpha_{max} - \alpha_{min}) \tag{2.1}$$

where each data point is entered as x, α_{min} is the smallest data value in the training and transfer sets, and α_{max} is the largest data value in the sets.

2.2 Back-propagation Parameters

A back-propagation network with 16 input nodes, 19 hidden nodes, and one output node was used for this study. Jando *et. al.* found this to be the optimal back-propagation network structure for this problem.

All of the initial network weights were randomly set to a value between -0.5 and 0.5. Some informal analysis was conducted to determine the largest acceptable value for the learning rate; a learning rate coefficient (λ) of .005 was used to obtain the results reported here.

The back-propagation formulas reported in section 1.2.4 were used, with slight modifications. Researchers have reported that the inclusion of bias nodes and momentum terms can improve network training performance (Freeman & Skapura, 1991; Caudill & Butler, 1990), and these features were included in the back-propagation network. The bias nodes are simply the addition of two input nodes (for the hidden and output layers) that always have an input value of 1. The inclusion of bias nodes does not change the formulas outlined in section 1.2.4.

The momentum term is intended to speed up network training by keeping network weight changes in the same general "direction" when a weight is updated. With the inclusion of a momentum term, the back-propagation equation 1.5 becomes:

$$w'_{mn} = (w'_{mn} + \lambda \cdot \beta_m) + \mu \cdot \delta^{old} \tag{2.2}$$

where λ is the learning coefficient, μ is the momentum term (set to .5 here), and δ^{old} is the previous change to the weight.

2.3 Predictive Autoencoder Parameters

A predictive autoencoder network with 16 input nodes, 19 hidden nodes, and 17 output nodes was used in order to keep the structure as close to the backpropagation network as possible. Similarly, the learning rate was set to .005, and the momentum term was assigned a value of .5. Again, all initial network weights were randomized between -.5 and .5.

2.4 Network Training

It has already been mentioned that the training and transfer sets were extracted from a larger 12-hour recording of time-series data from a single rat. Data patterns (vectors) consisting of 16 consecutive time-series recordings were presented as input to the 16 input nodes of the backpropagation and predictive autoencoder networks. In other words, vectors were created by using a moving window of 16 data points taken over the time series data.

The data file had been marked for the beginning and ending time-series point of all HVS events. If a vector contained 7 or more time-series points falling within a marked HVS event, the vector pattern was classified as an HVS vector.

During training of the backpropagation network, the desired output for the single output node was 1 for HVS vectors and 0 for non-HVS vectors. For 16 of the 17 predictive autoencoder output nodes, the desired outputs during training were the 16 network input values. The final output node (the classification node) had a desired output of 1 for HVS vectors and 0 for non-HVS vectors. The predictive autoencoder used the backpropagation algorithm to adjust weights during training.

Both networks were run on an IBM RS/6000 with 32-bit precision.

CHAPTER 3. RESULTS

3.1 Raw Data

Initially, the backpropagation network is slower in learning the two classifications (HVS vs. non-HVS) in raw EEG patterns. Figure 3.1 depicts the decrease of the summed squared error in classifying the training set after each training epoch. While the predictive autoencoder has approached its asymptote by approximately 15,000 epochs, it takes the backpropagation network significantly longer to reach the same reduction in summed squared error (approximately 20,000 epochs).

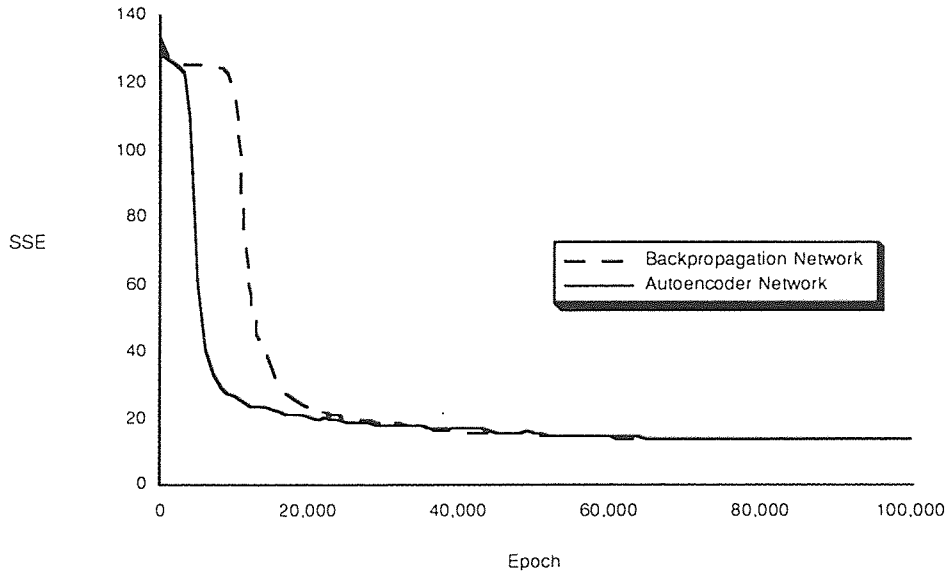

Figure 3.1 Training set SSE vs. training epoch

As the number of epochs increases, both the backpropagation and the predictive autoencoder networks begin to output values of 0 for non-HVS vectors and 1 for HVS vectors. The networks do especially well on the training set (figure 3.2a); however, the performance does not generalize as well to the transfer set (figure 3.2b).

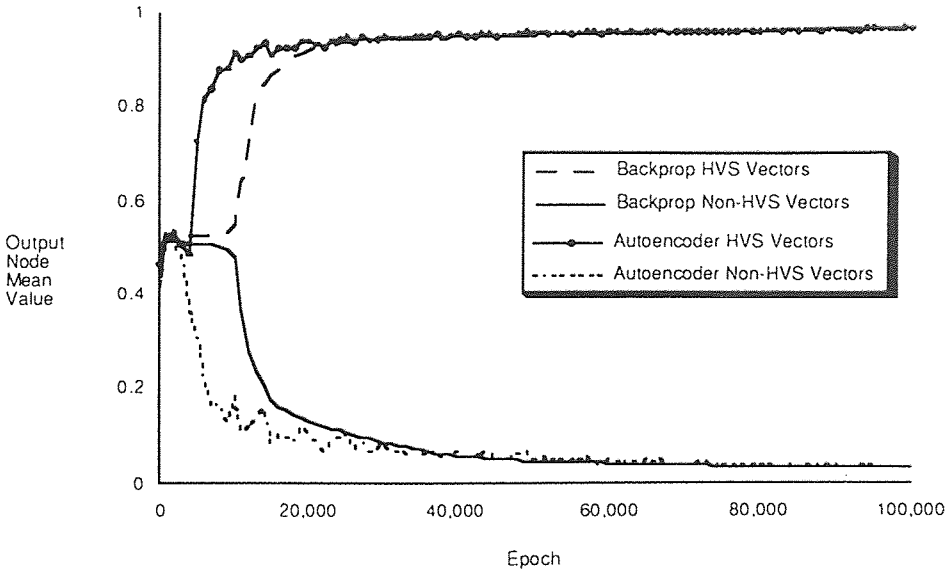

Figure 3.2a. Mean output value for training set vs. epoch.

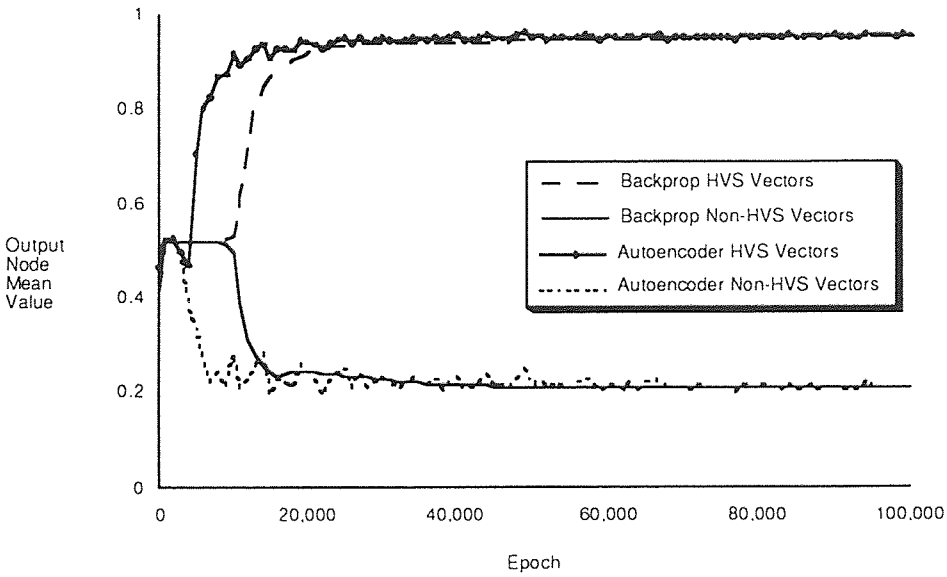

Figure 3.2b. Mean output value for transfer set vs. epoch.

Note that while non-HVS vectors in the training set generate outputs close to 0, output values generated by the non-HVS vectors in the transfer set remain around the value of .2. This is true for both networks.

As figure 3.1 shows, changes in the SSE became very small beyond 100,000 epochs. Training was stopped at 200,000 epochs, and performance on the transfer set was then tested. A threshold on the output (or classification) node was varied to determine the value which maximized correct classification of HVS and non-HVS vectors (figure 3.3).

With raw data, the back-propagation network could correctly classify at most 89% of both the HVS and non-HVS vectors (by setting the threshold close to .99). The predictive autoencoder could correctly classify at most 88% of the HVS and non-HVS vectors (by setting the threshold on the classification node close to .98).

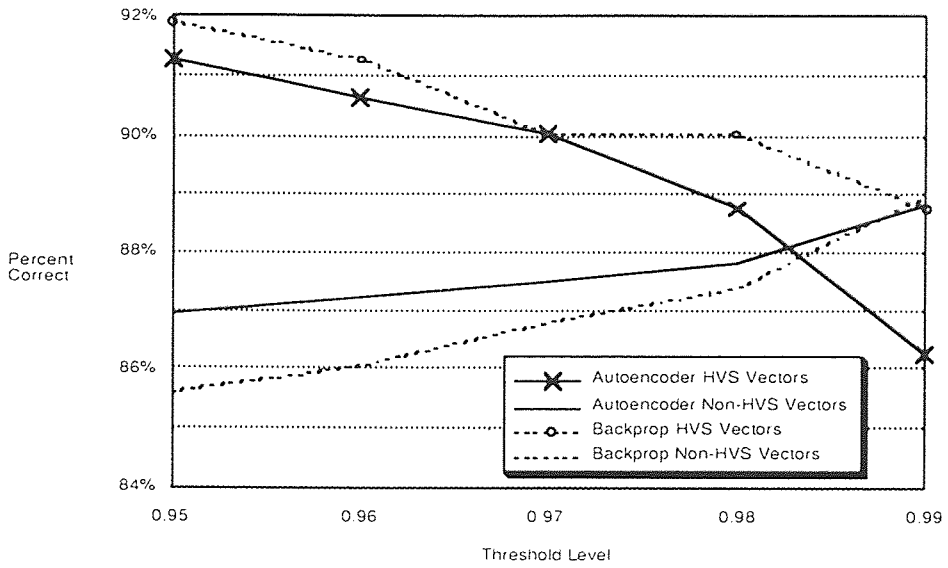

Figure 3.3. Results of varying the threshold level after training.

3.2 FFT Data

Results with the FFT data were similar to those found with the raw EEG data. For instance, figure 3.4 shows that the back-propagation network again needed more epochs before it began to make progress in adjusting network weights.

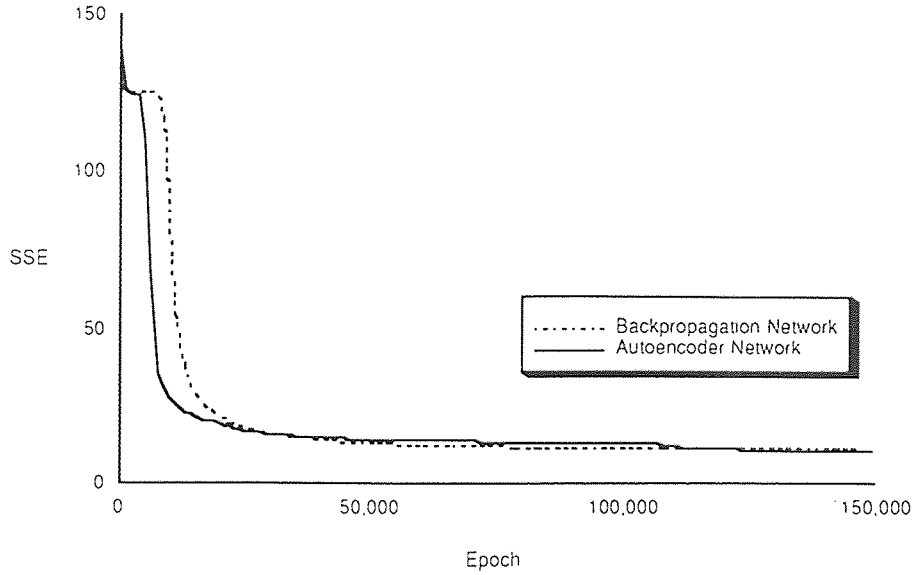

Figure 3.4. FFT training set SSE vs. epoch

During threshold testing with FFT data, back-propagation performance decreased to 87% correct classification of HVS and non-HVS vectors, while the predictive autoencoder improved to 89% correct classification. (See figure 3.5)

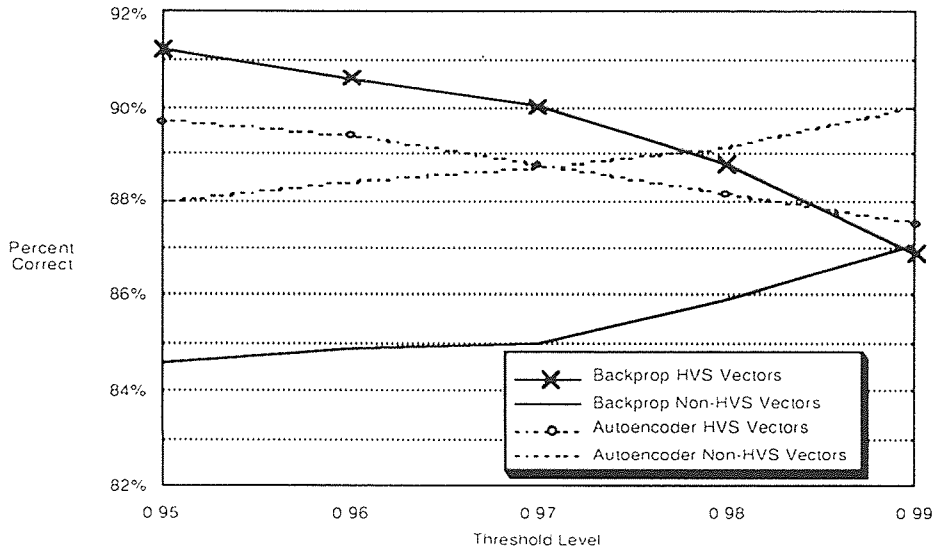

Figure 3.5. Results of varying the threshold level after FFT training.

CHAPTER 4. DISCUSSION

The results of this research do not indicate that one network is clearly superior to the other for the task of EEG classification. The predictive autoencoder network with FFT data sets and the backpropagation network with raw data sets displayed equivalent performance. However, there are several issues raised that merit further discussion.

4.1 Training Speed

Before the network simulations were run, it was believed that the predictive autoencoder would take longer to train. While the backpropagation network has only one output node, the predictive autoencoder has 17; more output nodes require more calculations to adjust their weights.

With both raw and FFT data, the predictive autoencoder required fewer epochs to start converging on an appropriate set of network weights. This suggests that by constraining the allowable weights, the predictive autoencoder can be useful in situations like EEG classification where there are a small number of network weight solutions that solve the problem.

In addition, interesting results related to training speed were observed when the FFT data sets were accidentally given to the networks without being normalized first. The backpropagation network learned extremely quickly, although with poor generalization to the transfer set. In contrast, the predictive autoencoder had an extremely hard time learning any patterns at all. The network was attempting to find weights that would recreate input patterns outside the range of 0-1, but the sigmoid output function would never let the nodes achieve this goal.

4.2 Resistance to Overtraining

As training was continued with the FFT training set out to 200,000 epochs, the backpropagation network started to overtrain. That is, while performance on the training set was still increasing, the classification performance on the transfer set was deteriorating. The predictive autoencoder did not display any characteristics of overtraining under any condition. This suggests that the predictive autoencoder may be more resistant to overtraining then the backpropagation network.

4.3 "Noise" in Predictive Autoencoder Graphs

The graphs of output node mean value using the raw data (figure 3.2) show that backpropagation results display less variability than those for the predictive autoencoder. The results with FFT data sets were almost identical.

In retrospect, this finding can be easily explained. The predictive autoencoder has 17 output nodes instead of one, and any of those output nodes can have a large influence on adjusting the network weights. If the output node with the largest error during an epoch is not the classification node, the graph of the classification node performance will show the variability seen in figure 3.2.

The practical implication of this is that it is necessary to train the predictive autoencoder over a sufficient number of epochs. If training is interrupted before the SSE has decreased sufficiently, this variability in classification might result in poor network performance (compared to a backpropagation network trained over the same number of epochs).

4.4 FFT Performance Deterioration

While Jando *et. al.* (1993) reported that the fast Fourier transform could improve results significantly, we found that the transformation could actually lead to worse classification performance using our limited training sets. Given the constraints imposed by the real world, it is common to find neural networks being trained with limited training sets; the results reported here suggest that a comparison of FFT vs. raw data should always be conducted for performance comparison.

4.5 Suggestions for Future Research

Many of the preliminary results reported here could be investigated in more depth. For example, only one predictive autoencoder structure was tested. Future research could explore how varying the predictive autoencoder (e.g., fewer hidden nodes, etc.) affects classification performance.

Studies could be designed to specifically test the resistance of the predictive autoencoder to overtraining. Another area of research (of general interest to all neural network researchers) would explore the conditions under which the FFT leads to decreased performance.

Finally, future research should explore other neural network architectures. Of particular interest are networks using unsupervised learning, since it is a tedious job to mark 12 hour EEG recordings for epileptic HVS events which last approximately four seconds.

APPENDICES

A. Back-propagation Training Code

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

/*
        C Program Implementing BackPropigation
        written by Brian Armieri, January 1993

        This program reads in the EEG data from the integer file
        DATA_FILE designated below.  The data file is
        expected to consists of 3000 vectors of 16 data points each.
        That is, 16 numbers are read in and assigned to 1 vector, then
        the next 16 are read...

        The first 512 vectors are used as training vectors, and the last
        1000 vectors are used as transfer vectors.

        For all vectors, the desired output has been "hard wired" into the
        program.  Procedure read_data (see below) associates vectors
        numbered between 60 and 326 and vectors numbered between 2476
        and 2796 with a desired output value of 1.  All others have a desired
        output of 0.  These numbers are specific to the data file data.set.

        Results are saved every epoch in the file RESULTS designated
        below.  The network weights are also saved in the file WEIGHTS.

*/

#define DATA_FILE "/u/armieri/NNdir/Raw_data/data.set"
#define RESULTS "/u/armieri/NNdir/Backprop/results"
#define WEIGHTS "/u/armieri/NNdir/Backprop/weights"
#define NEW_WEIGHTS "/u/armieri/NNdir/Backprop/new_weights"

/* constants */
#define NUM_INPUT   16
#define NUM_HIDDEN  19
#define NUM_OUTPUT  1
```

```
/* Macros */
#define square(x)       ((x)*(x))
#define sigmoid(x)      (1.0/(1.0 + exp((double)-(x))))
#define sigDeriv(x)     ((x)*(1.0-(x)))

/* Data Structures */
typedef struct unit /* data structure for a single node */
{ double output;
 double target;
 double delta; } Unit;

typedef struct network /* data stucture for a 3-layer network */
{double input[NUM_INPUT];
 Unit hidden[NUM_HIDDEN];
 Unit output[NUM_OUTPUT];
 double i2h_weight[NUM_INPUT][NUM_HIDDEN];
 double i2h_momentum[NUM_INPUT][NUM_HIDDEN];
 double h2o_weight[NUM_HIDDEN][NUM_OUTPUT];
 double h2o_momentum[NUM_HIDDEN][NUM_OUTPUT];
 double b2h_weight[NUM_HIDDEN];
 double b2h_momentum[NUM_HIDDEN];
 double b2o_weight[NUM_OUTPUT];
 double b2o_momentum[NUM_OUTPUT];
 double error;} Network;

/* Global variables for network calculations */
Network  EEGnet;
double   Learning_Rate=0.005;
double    momentum_parameter=.5;
double   Threshold=.85;
double   ESE=0.0; /* event standard error */
double   NESE=0.0; /* non-event stand. err */
double   TESE=0.0; /* Training HVS event standard error */
double   TNESE=0.0; /* training non-event stand. err. */
double   NEMO = 0.0; /* non-event mean output */
double   EMO  = 0.0; /* event mean-output */
double   Input_Value[3000][16];
double   Train_Vector[512][16];
int      Target_Value[3000];
unsigned long int epoch=0;
```

```
/* Main Program */
main()
{ int counter,counter2;
  FILE *write_to;

  /* load weights */
  write_to = fopen(WEIGHTS,"r");
  for (counter=0;counter<NUM_INPUT;counter++)
    for (counter2=0;counter2<NUM_HIDDEN;counter2++)
      fscanf(write_to,"%lf\n",&EEGnet.i2h_weight[counter][counter2]);
  for (counter=0;counter<NUM_HIDDEN;counter++)
      fscanf(write_to,"%lf\n",&EEGnet.b2h_weight[counter]);
  for (counter=0;counter<NUM_HIDDEN;counter++)
    for (counter2=0;counter2<NUM_OUTPUT;counter2++)
      fscanf(write_to,"%lf\n",&EEGnet.h2o_weight[counter][counter2]);
  for (counter=0;counter<NUM_OUTPUT;counter++)
      fscanf(write_to,"%lf\n",&EEGnet.b2o_weight[counter]);
  fclose(write_to);

  /* initialize momentum weights to zero */
  for (counter=0;counter<NUM_INPUT;counter++)
    for (counter2=0;counter2<NUM_HIDDEN;counter2++)
     EEGnet.i2h_momentum[counter][counter2] = 0;
  for (counter=0;counter<NUM_HIDDEN;counter++)
   {EEGnet.b2h_momentum[counter] = 0;
    for (counter2=0;counter2<NUM_OUTPUT;counter2++)
    { EEGnet.h2o_momentum[counter][counter2] = 0;
     EEGnet.b2o_momentum[counter2] = 0; } }

  /* create output file */
  write_to = fopen(RESULTS,"w");

fprintf(write_to,"epoch,TOK,TEMO,TESE,TNEMO,TNESE,TSSE,EMO,ESE,N
EMO,NESE\n");
  fclose(write_to);

  /* get data from data file */
  read_data();

  /* perform network calculations */
  train_network();
}
```

```
train_network()
{ int num_data_points, temp, counter, counter2, counter3;
  FILE *write_to;

  get_results(); /* write initial results with rand weights */

  /* run training for (num_data_points*counter) epochs */
   for (num_data_points=1;num_data_points<200;num_data_points++)
   {
    /* counter, below, determines # of epochs between writing of results */
    for (counter=0;counter<1000;counter++)
     { epoch++;
       for (counter2=0;counter2<512;counter2++) /* choose 512 vectors */
        { temp = (rand() & 511); /* choose a random training vector */
          /* put random vector as network input */
          for (counter3=0;counter3<16;counter3++)
           EEGnet.input[counter3] = Train_Vector[temp][counter3];
          calculate_hidden_layer();
          calculate_output_layer();
          /* set desired value for the network output */
         EEGnet.output[0].target = 0;
          if ((temp>59)&&(temp<326)) EEGnet.output[0].target = 1;
          /* adjust weights */
          backprop_weights();
        }
     } /* end for counter */
   get_results(); /* see how network does with new weights */
   } /* end for num_of_data_points */
} /* end procedure */

/* procedure to test training & transfer sets with current net weights */
/*   this procedure writes results to an output file */
get_results()
{ int counter, counter2;
 int OK=0;
 double TSSE=0;
 double TEMO=0;
 double TNEMO=0;
 double TESE = 0.0;
 double TNESE = 0.0;
 FILE *write_to;
```

```
/* test all training vectors */
  for (counter=0;counter<512;counter++)
   {
    for (counter2=0;counter2<16;counter2++)
     EEGnet.input[counter2] = Train_Vector[counter][counter2];
    calculate_hidden_layer();
    calculate_output_layer();
   if ((counter<326) && (counter>59))
    { /* begin if */
     if (EEGnet.output[0].output >= Threshold) OK++;
     TSSE += (square(1-EEGnet.output[0].output));
     TEMO += EEGnet.output[0].output;
    } /* end if */
   else
    { /* begin else */
     if (EEGnet.output[0].output < Threshold) OK++;
     TSSE += (square((EEGnet.output[0].output)));
     TNEMO += EEGnet.output[0].output;
    } /* end else */
  } /* end for */
 TEMO = (TEMO / 266);
 TNEMO = (TNEMO / 246);

 /* calculate standard error for training vectors */
 /* standard error = (sqrt(summation(actual-expected)) / N) */
 for (counter=0;counter<512;counter++)
  { for (counter2=0;counter2<16;counter2++)
      EEGnet.input[counter2] = Input_Value[counter][counter2];
    calculate_hidden_layer();
    calculate_output_layer();
   if ((counter<326) && (counter>59))
     TESE += (square(EEGnet.output[0].output-TEMO));
   else
     TNESE += (square(EEGnet.output[0].output-TNEMO));
 } /* end for counter */
 TESE = ((sqrt(TESE))/266);
 TNESE = ((sqrt(TNESE))/246);


  test_transfer();  /* call procedure to test transfer set */
    write_to = fopen(RESULTS,"a"); /* write results to results file */

fprintf(write_to,"%d,%d,%6.3lf,%6.3lf,%6.3lf,%6.3lf,%6.3lf,%6.3lf,%6.3lf,%6.3
lf,%6.3lf\n",
        epoch,OK,TEMO,TESE,TNEMO,TNESE,TSSE,EMO, ESE,NEMO,NESE);
  fclose(write_to);
```

```
  write_to = fopen(NEW_WEIGHTS,"w");
  for (counter=0;counter<NUM_INPUT;counter++)
    for (counter2=0;counter2<NUM_HIDDEN;counter2++)
      fprintf(write_to,"%lf\n",EEGnet.i2h_weight[counter][counter2]);
  for (counter=0;counter<NUM_HIDDEN;counter++)
    fprintf(write_to,"%lf\n",EEGnet.b2h_weight[counter]);
  for (counter=0;counter<NUM_HIDDEN;counter++)
    for (counter2=0;counter2<NUM_OUTPUT;counter2++)
      fprintf(write_to,"%lf\n",EEGnet.h2o_weight[counter][counter2]);
  for (counter=0;counter<NUM_OUTPUT;counter++)
    fprintf(write_to,"%lf\n",EEGnet.b2o_weight[counter]);
 fclose(write_to);
}


/* Function to test transfer set with current network weights */
/*  Results are all stored in global variables */
test_transfer()
{ int counter,counter2;
 int temp=0;

 ESE = 0.0;
 NESE = 0.0;
 EMO = 0;
 NEMO = 0;

 for (counter=2000;counter<3000;counter++) /* try 1000 data vectors */
 { for (counter2=0;counter2<16;counter2++)
    EEGnet.input[counter2] = Input_Value[counter][counter2];
   calculate_hidden_layer();
   calculate_output_layer();
   /* add output to sum(HVS vector outputs) or sum(non-HVS vector
outputs) */
   /* note EMO right now holds sum(HVS vector outputs) */
   /* note NEMO right now hols sum(non-event vector ouputs) */
   if (Target_Value[counter] == 1) EMO += EEGnet.output[0].output;
   else NEMO += EEGnet.output[0].output;
 } /* end for */
 EMO = EMO / 320; /* divide for final Event Mean Output */
 NEMO = NEMO / 680; /* divide for final Non-Event Mean Output */
```

```
 /* calculate standard errors */
  for (counter=2000;counter<3000;counter++)
   {
    for (counter2=0;counter2<16;counter2++)
     EEGnet.input[counter2] = Input_Value[counter][counter2];
    calculate_hidden_layer();
    calculate_output_layer();
    if (Target_Value[counter] == 1)
     ESE += square(EEGnet.output[0].output - EMO);
    else
     NESE += square(EEGnet.output[0].output - NEMO);
   } /* end for counter */
  ESE = ((sqrt(ESE))/320);
  NESE = ((sqrt(NESE))/680);

} /* end get_results */


/* Function to calculate hidden layer's input and output */
calculate_hidden_layer()
{ double result;
  int counter1, counter2;

   for (counter1=0;counter1<NUM_HIDDEN;counter1++)
    { result = 0.0; /* clear value of result */

     /* calc summation of (inputs*input weights) */
      for (counter2=0;counter2<NUM_INPUT;counter2++)
       result += (EEGnet.input[counter2] *
              EEGnet.i2h_weight[counter2][counter1]);
     result += EEGnet.b2h_weight[counter1];
     result = sigmoid(result); /* apply sigmoid for output level */

     EEGnet.hidden[counter1].output = result; }
}
```

```
/* Function to calculate output layer's input and output */
calculate_output_layer()
{ double result;
  int dummy_var_1, dummy_var_2;

  for
(dummy_var_1=0;dummy_var_1<NUM_OUTPUT;dummy_var_1++)
    { result = 0.0; /* clear value of result */

     /* calc summation of (inputs*input weights) */
    for
(dummy_var_2=0;dummy_var_2<NUM_HIDDEN;dummy_var_2++)
       result += (EEGnet.hidden[dummy_var_2].output *
              EEGnet.h2o_weight[dummy_var_2][dummy_var_1]);
     result += EEGnet.b2o_weight[dummy_var_1]; /* include bias term */
     result = sigmoid(result);

     EEGnet.output[dummy_var_1].output = result; }
}


/* Function to use backpropagation learning rule */
backprop_weights()
{ int counter, counter2;
  double temp;

 EEGnet.output[0].delta =
    ( (EEGnet.output[0].target - EEGnet.output[0].output) *
     (sigDeriv(EEGnet.output[0].output)) * Learning_Rate  );

  /* calc hidden node errors */
  for (counter=0;counter<NUM_HIDDEN;counter++)
   { EEGnet.hidden[counter].delta =
        ( (sigDeriv(EEGnet.hidden[counter].output)) *
         EEGnet.output[0].delta * EEGnet.h2o_weight[counter][0] );

   /* adjust weights on inputs to hidden */
    for (counter2=0;counter2<NUM_INPUT;counter2++)
     { EEGnet.i2h_momentum[counter2][counter] =
         ((EEGnet.hidden[counter].delta * EEGnet.input[counter2]) +
          (momentum_parameter *
EEGnet.i2h_momentum[counter2][counter]));
       EEGnet.i2h_weight[counter2][counter] +=
          EEGnet.i2h_momentum[counter2][counter]; }
```

```
/* adjust weight on bias to hidden */
    EEGnet.b2h_momentum[counter] =
      ( EEGnet.hidden[counter].delta +
          (momentum_parameter * EEGnet.b2h_momentum[counter]));
    EEGnet.b2h_weight[counter] += EEGnet.b2h_momentum[counter];
   }

   /* update weights to output bias */
    EEGnet.b2o_momentum[0] = ( EEGnet.output[0].delta +
                        (momentum_parameter * EEGnet.b2o_momentum[0]));
    EEGnet.b2o_weight[0] += EEGnet.b2o_momentum[0];

   /* update weights to output nodes */
    temp = EEGnet.output[0].delta;
     for (counter2=0; counter2<NUM_HIDDEN;counter2++)
      { EEGnet.h2o_momentum[counter2][0] =
          ((temp * EEGnet.hidden[counter2].output) +
            (momentum_parameter * EEGnet.h2o_momentum[counter2][0]));
       EEGnet.h2o_weight[counter2][0] +=
          EEGnet.h2o_momentum[counter2][0]; }
}


/* Function to read data into program */
read_data()
{ FILE *EEGdata;
  int num,t1,t2,counter,counter2,target;

   if ((EEGdata = fopen(DATA_FILE,"r"))==0)
     { printf("\nError opening data file, execution halted.\n");
     exit(1); }
  target = 0;
   for (counter=0;counter<3000;counter++)
     { for (counter2=0;counter2<16;counter2++)
        { fscanf(EEGdata,"%d %d %d\n",&t1,&t2,&num);
         Input_Value[counter][counter2] = (double)(num-1576)/1047;  }
      /* set desired network output associated with the vector just read */
     if ((counter==60)||(counter==2476))  target = 1;
     if ((counter==326)||(counter==2796)) target = 0;
      Target_Value[counter] = target; }
  fclose(EEGdata);
  for (counter=0;counter<512;counter++)
    for (counter2=0;counter2<16;counter2++)
       Train_Vector[counter][counter2] = Input_Value[counter][counter2];}
```

B. Predictive Autoencoder Training Code

The C code for the predictive autoencoder was similar to that for the back-propagation network. The only procedure that required significant modification was **backprop_weights**, shown below:

```
/* Function to use backpropagation learning rule */
backprop_weights()
{ int counter, counter2;
  double temp;

  /* Hippocampal model addition to backprop network */
  for (counter=0;counter<16;counter++)
   EEGnet.output[(counter+1)].target = EEGnet.input[counter];

  for (counter=0; counter<NUM_OUTPUT;counter++)
   EEGnet.output[counter].delta =
     (EEGnet.output[counter].target - EEGnet.output[counter].output) *
     sigDeriv(EEGnet.output[counter].output);

  /* calc hidden node errors */
  for (counter=0;counter<NUM_HIDDEN;counter++)
   {
   EEGnet.hidden[counter].delta = 0;
    for (counter2=0;counter2<NUM_OUTPUT; counter2++)
     EEGnet.hidden[counter].delta .+=
      (EEGnet.output[counter2].delta*EEGnet.h2o_weight[counter][counter2]);
   EEGnet.hidden[counter].delta *=
sigDeriv(EEGnet.hidden[counter].output);
    EEGnet.hidden[counter].delta *= Learning_Rate;

   /* adjust weights on inputs to hidden */
    for (counter2=0;counter2<NUM_INPUT;counter2++)
     { EEGnet.i2h_momentum[counter2][counter] =
         ((EEGnet.hidden[counter].delta * EEGnet.input[counter2]) +
          (momentum_parameter *
EEGnet.i2h_momentum[counter2][counter]));
      EEGnet.i2h_weight[counter2][counter] +=
         EEGnet.i2h_momentum[counter2][counter]; }
```

```
/* adjust weight on bias to hidden */
    EEGnet.b2h_momentum[counter] =
      ( EEGnet.hidden[counter].delta +
          (momentum_parameter * EEGnet.b2h_momentum[counter]));
    EEGnet.b2h_weight[counter] += EEGnet.b2h_momentum[counter];
  }

  /* update weights to output nodes */
    for (counter=0;counter<NUM_OUTPUT;counter++)
    {
     temp = EEGnet.output[counter].delta * Learning_Rate;

     EEGnet.b2o_momentum[counter] = ( temp +
                  (momentum_parameter * EEGnet.b2o_momentum[counter]));
     EEGnet.b2o_weight[counter] += EEGnet.b2o_momentum[counter];

     for (counter2=0; counter2<NUM_HIDDEN;counter2++)
     {
      EEGnet.h2o_momentum[counter2][counter] =
        ((temp * EEGnet.hidden[counter2].output) +
          (momentum_parameter *
EEGnet.h2o_momentum[counter2][counter]));
      EEGnet.h2o_weight[counter2][counter] +=
          EEGnet.h2o_momentum[counter2][counter];
     } /* end for counter2 */
    } /* end for counter */

     } /* end procedure */
```

REFERENCES

Caudill, M. and Butler, C. 1990. *Naturally Intelligent Systems.* Cambridge, MA: MIT Press.

Freeman, J. and Skapura, D. 1991. *Neural Networks: Algorithms, Applications, and Programming Techniques.* Reading, MA: Addison-Wesley Publishing Company.

Gluck, M. and Myers, C. 1992. Hippocampal-System Function in Stimulus Representation and Generalization: A Computational Theory. *Proceedings of the Fourteenth Annual Meeting of the Cognitive Science Society.* Hillsdale, NJ: Erlbaum Associates.

Gluck, M. and Myers, C. 1993. Hippocampal Mediation of Stimulus Representation: A Computational Theory. *Hippocampus.* In press.

Hinton, G. 1989. Connectionist Learning Procedures. *Artificial Intelligence,* **40**: 185-234.

Hinton, G. 1992. How Neural Networks Learn From Experience. *Scientific American,* **266**: 145-151.

Jando, G., Siegel, R., Horvath, Z., and Buzsaki, G. 1993. Pattern Recognition of the Electroencephalogram by Artificial Neural Networks. *Electroencephalography and Clinical Neurophysiology,* **86**: 100-109.

McClelland, J. and Rumelhart, D. 1986. *Parallel Distributed Processing, volumes 1 and 2.* Cambridge, MA: MIT Press.