Yale University
Department of Computer Science

Selections of Term Projects for CS426 & 477
1995 & 1996

Instructor: W. Miranker

YALEU/DCS/TR-1127

Contents

Preface

This is a selection of term projects from the courses on neural nets and applications in the fall semester of both '95 and '96. Although this is but a sample of possibilities, the scope of the effectiveness of neural nets in application is certainly conveyed. We hope to add to this collection of reports in the future, augmenting both the range and the depth of the work.

I am grateful to my teaching assistant, Hong Xiao for her help in pulling this together.

W. Miranker

Neural Network in Market Predictions

Ming Deng
Department of Physics, Yale University, New Haven,CT 06520-8120

Qiang Li
Department of Statistics, Yale University, New Haven, CT 06511

ABSTRACT

Standard Back-Propagation neural networks are implemented to predict the market prices of equities. Both industrial indices and historical data of the stock are used as training information to train the neural network until it converges. To minimize the delay in the response of the network, day to day changes are also used as training data. The performance of the neural network is evaluated.

1. Introduction

The stocks and equity markets have long been known to be systems having random walk behaviors. The question of making predictions of the market behavior has been a fascinating, albeit impossible problem. Various statistical methods have been devised in an effort to model the market behaviors. However, when the numerous factors in the real life come in, it is hard for these models to adjust fast enough to follow market trends that depend on the political and economical scene.

Multilayer perceptron neural networks have long been used successfully in pattern recognition and memory simulation. If the neural network can be trained to memorize the market behavior or its patterns, it may provide a powerful tool in understanding the market trends. The adaptive behavior of the neural network makes this task very promising.

2. Implementation and Results

2.1. Implementation of Back-Propagation Neural Network

Our implementation uses the multilayer perceptrons by training them using the Back Propagation algorithm. The algorithm is a generalization of the more ubiquitous least-mean-square (LMS) algorithm. The advantages of the multilayer perceptrons are its distinct characteristics of nonlinearity and high degree of connectivity. Such neural networks have been used successfully in the problem of pattern recognition. The patterns and features are extracted progressively by the internal neurons which have logistic function as its activation function.

Standard back propagation neural networks are coded in C. To study the behavior of the neural network and achieve fast convergence, the program is implemented to allow the total number of layers and the number of neurons in each layer be adjusted. Although previous studies in homework problems of CS477 have shown that the large number of neurons does not necessarily lead to faster convergence or better performances.

Finally, a simple three layer perceptron is used since tests show that going to larger and more sophisticated nets does not give that much improvements in performance. The input and output layers consist of linear neurons, and the hidden layer uses nonlinear sigmoidal neurons. The input layer and output layers typically have ~ 10 neurons. They are fully connected, which makes the total number of connections in the neural network to be ~ 100 .

2.2. Training Data Set

The historical data of stocks and indices are obtained from Datastream in the Social Science Library. The data encompasses the first 240 days of 1996. The indices used are *NYSE composite, Dow Joans composite, S&P 500, and NASDAQ.* The stock data are those of *Intel, Microsoft, Oracle, Sun Microsystems, Hewlett-Packard, Netscape Communications, Micron Electronics, Texas Instruments and Motorola.* Most of these are high-tech companies which have enjoyed significant growth during the past 1996 fiscal year.

The training data are 150 days of historical data of a particular stock and the industrial indices during that period of time. At first, the inputs are specified to be the prices of the stock in a consecutive period of 7 days and the industrial composites on 7th day, and the output used in the training is the price on the 8th day. To allow equal influence of different factors, all the indices and prices are normalized to have the same average. Such normalization is necessary because the sigmoidal function varies only in a small range near zero.

The training data sets are selected randomly in order to minimize a tendency in memorizing more recent effects. Each training set of data is run through the network in 100 epochs to make the network converge. To accelerate the convergence of the neural network, a momentum term (Chapter 6.3, Simon Haykin) is included to increase the rate of learning.

After the neural network is trained through these epochs, the converged weights are ready to be used to predict the stock behavior. The resulting neural network is then used to predict stock prices for all the 230 days of data we have downloaded.

2.3. Results

The program is run for each of the 9 stocks we have selected. And three of the typical result graphs are shown in Figure 1-3. They are the stock prices of *Microsoft, Micron Electronics and Motorola.*

The Microsoft stock prices shows a general trend of increase marked by a few little oscillation in short time scales. As shown in the graph, the neural net followed the trend of market very nicely. More over, it is not sensitive to the local fluctuations which tend to be kind of like noise. However, detailed examination shows that the neural network shows a delay to larger changes in the market. Basically, the neural network does not predict very well large changes in the market in short time scales.

The Micron stock and Motorola prices show great variation during the past year. As shown in the Figure 1-3, the neural network followed the market trend and the time delay in predicting the market change seems to be much smaller than the Microsoft graph, which means that the neural network did better in predicting upcoming changes of the market. This might because they are more similar to random walk behaviors and statistically have more extractable patterns in their behavior.

To study the pattern of changes in the stock market, stock price changes on consecutive 7 day period are plotted in two groups in Figure 4. To show their distribution, histogram are also made in Figure 5 and 6. The first group corresponds to the 7 day periods which is followed by a growth of the stock price. The second group corresponds to the 7 day periods which is followed by a drop in the stock price. In this way, the different historical pattern that leads to future changes can be studied.

The plots do show some interesting patterns. The group that is followed by growth has daily changes lying between -10.0 and 3.0, most of which are in the range of -2.0 and 1.0. Whereas the group that is followed by drop has daily changes lying between -8.0 and 3.0, most of which are in the range of -1.0 and 2.0. The distributions also show drastic difference for the two cases plotted in Fig 5 and 6. Therefore, the conclusion is that there may be some patterns in historical data that can hopefully lead to predicting market changes.

If the market behavior can be modeled by a random walk, then the above result can be explained. Since the probability of the market to rise and fall on the day by day basis are roughly the same, it is unlikely to have consecutive increases and decreased for very long. A long increase is more likely to be followed by a decrease in the market, and the same holds reversely.

In light of the above result, the program is modified to predict the market changes rather than the market price itself. Both the inputs and outputs are changed to be the day-to-day changes in the prices and indices. Equivalently, the derivatives of stocks and indices to time in days are used. The results are plotted in Figure 7-9 for the same three stock prices. The delay and difference between the market price and predicted price is smaller, however, the delay still exists.

3. Conclusion

Neural networks are implemented in an attempt to predict stock market prices. The network converges and seems to do very well in memorizing market behaviors on long term time scales. However, the neural network does not do very well in predicting sharp rises and falls of the market which may be due to random effects. Efforts has also been made to identify possible historical patterns, the result shows some improvement once the day-to-day changes are used. Further research is needed to understand these patterns.

REFERENCES

David M. Skapura, Building Neural Networks, ACM Press Books

Simon Haykin, Neural Networks, Macmillan Publishing Books

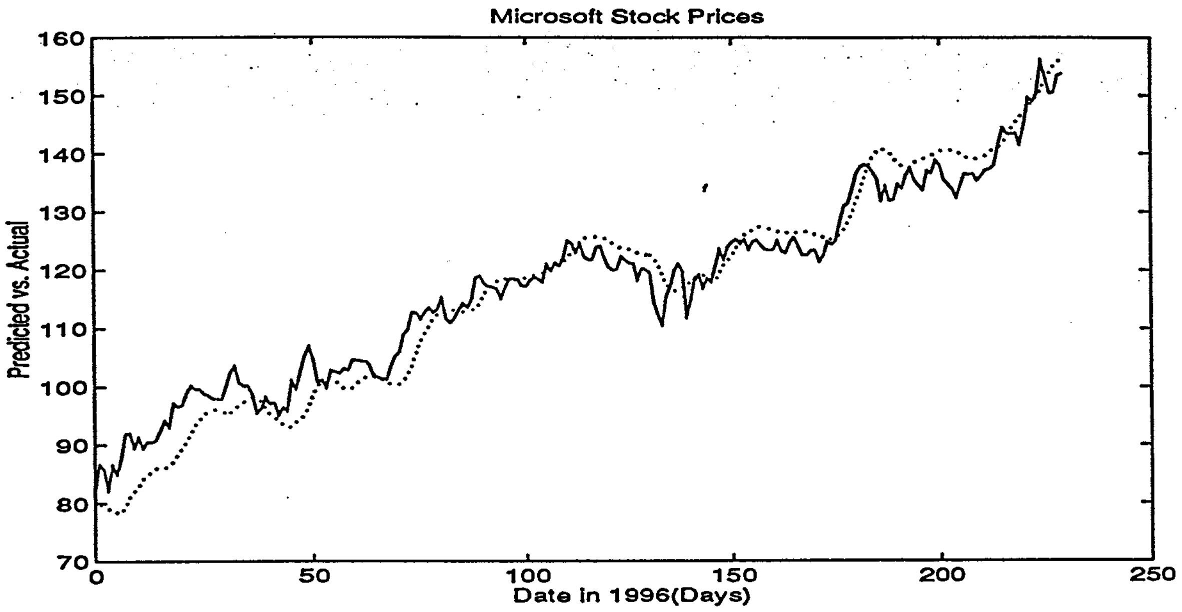


Fig. 1.— The predicted vs. actual stock prices of Microsoft in 1996. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

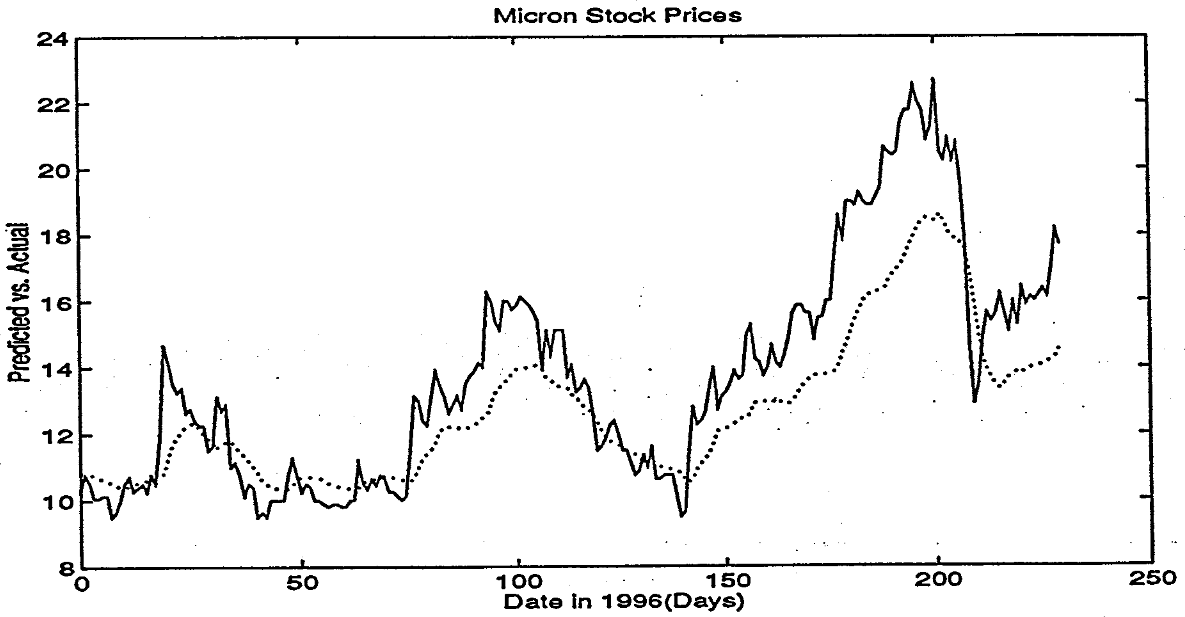


Fig. 2.— The predicted vs. actual stock prices of Micron in 1996. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

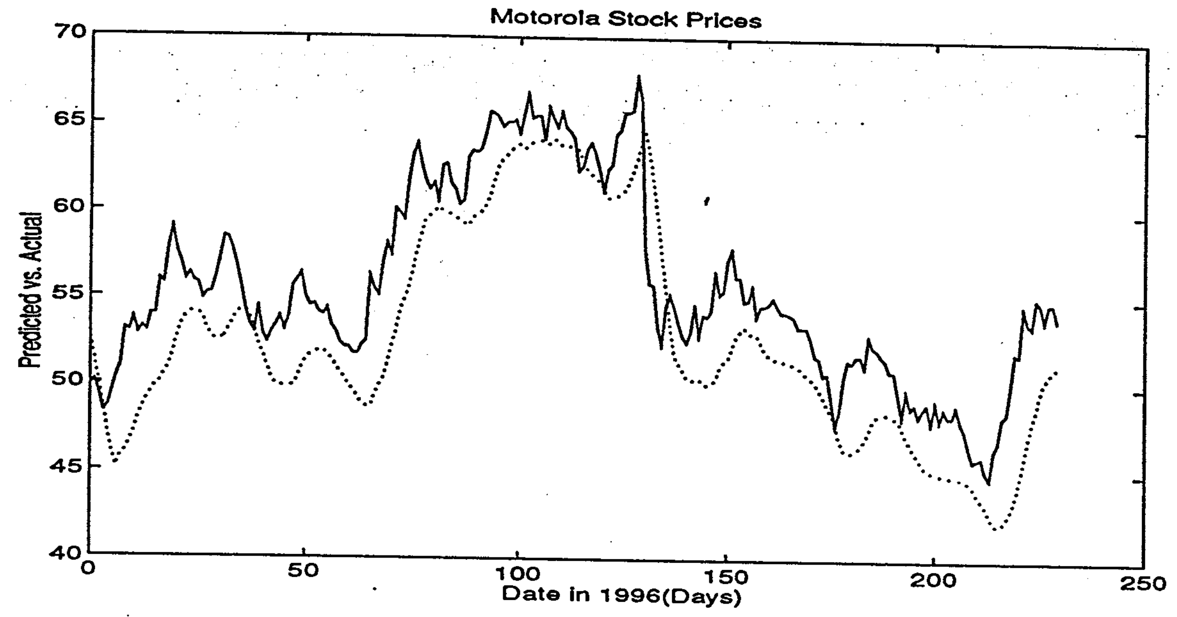


Fig. 3.— The predicted vs. actual stock prices of Motorola in 1996. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

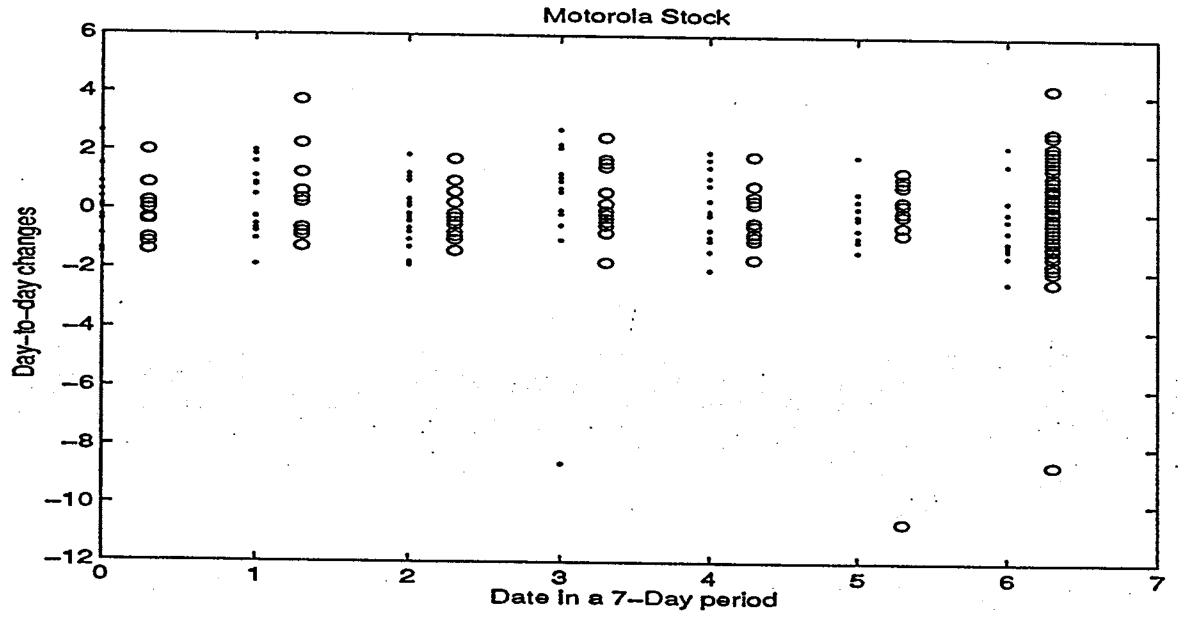


Fig. 4.— The daily changes of Motorola stock on consecutive 7 days. The dots correspond to 7 daily changes before price falls; the circles correspond to 7 daily changes before price rises.

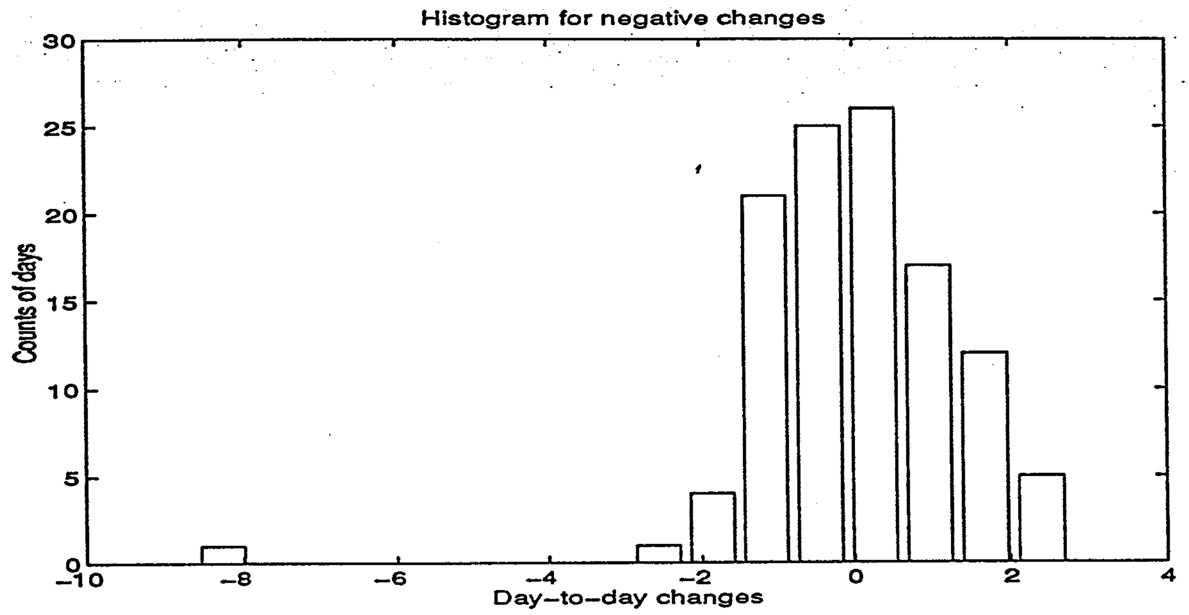


Fig. 5.— The histogram of daily changes before stock price falls.

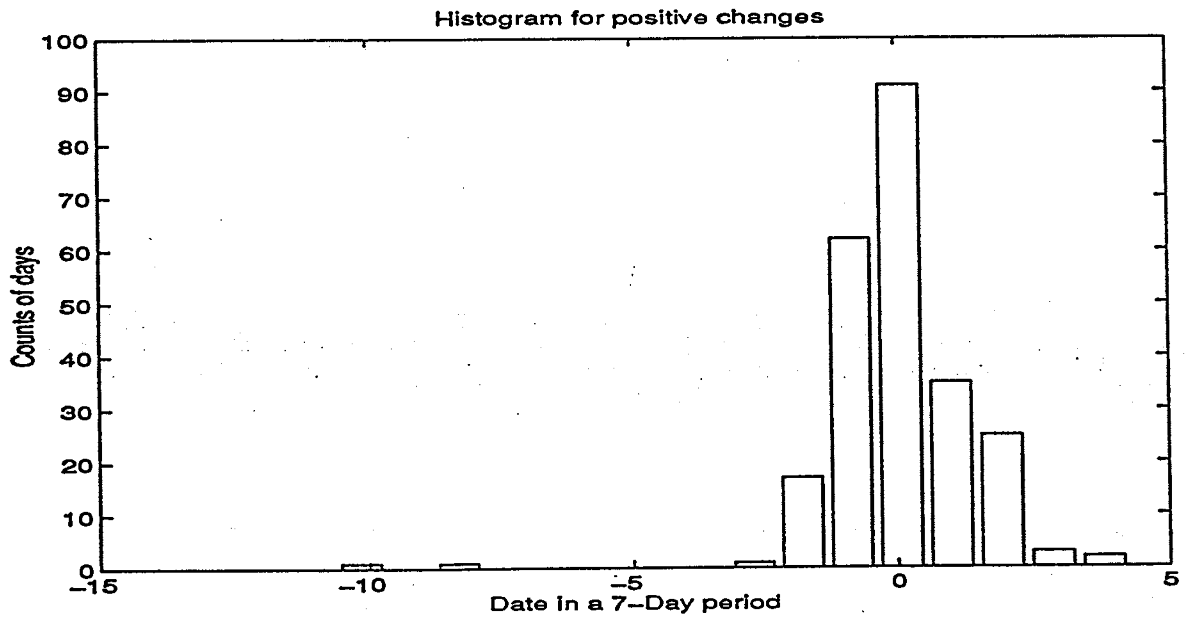


Fig. 6.— The histogram of daily changes before stock price rises.

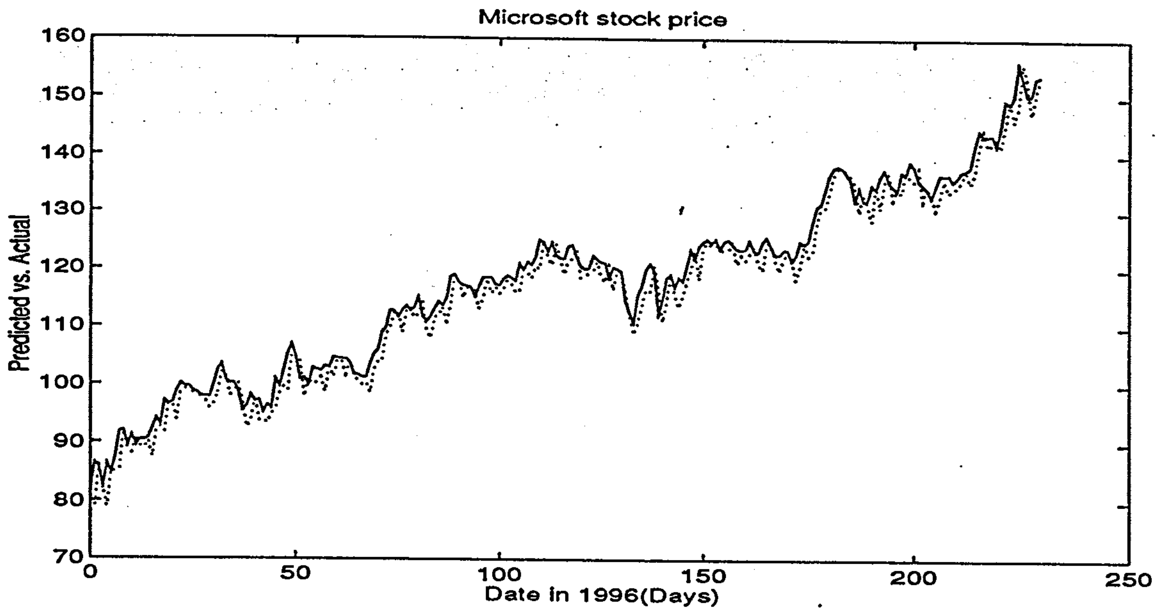


Fig. 7.— The predicted vs. actual stock prices of Microsoft in 1996. The method is based on daily changes rather than prices. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

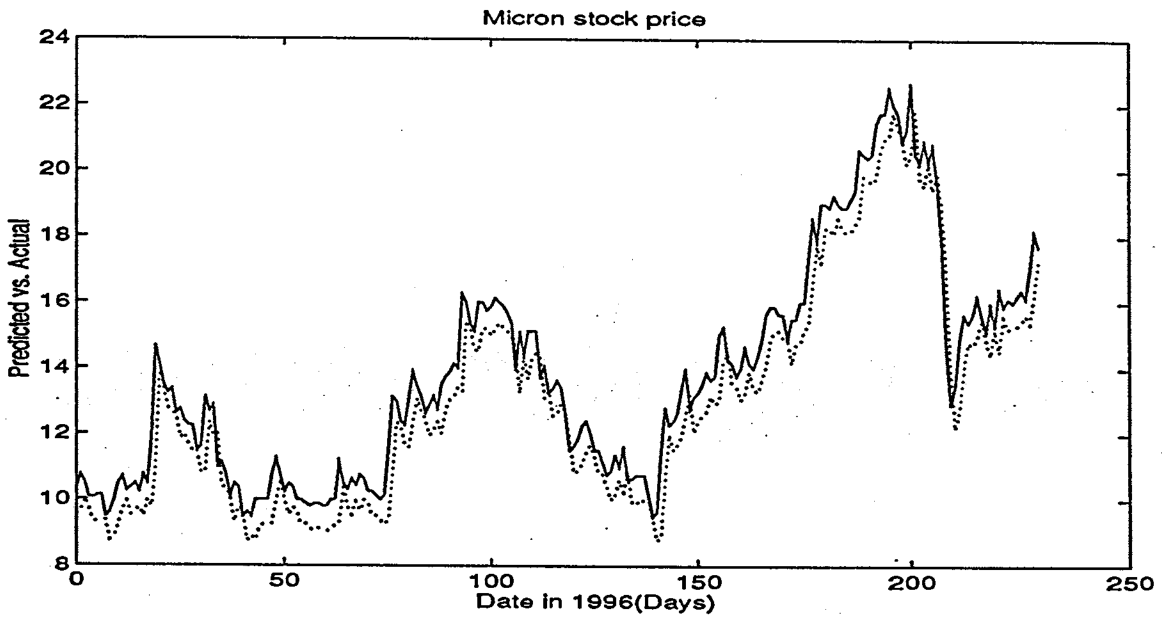


Fig. 8.— The predicted vs. actual stock prices of Micron in 1996. The method is based on daily changes rather than prices. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

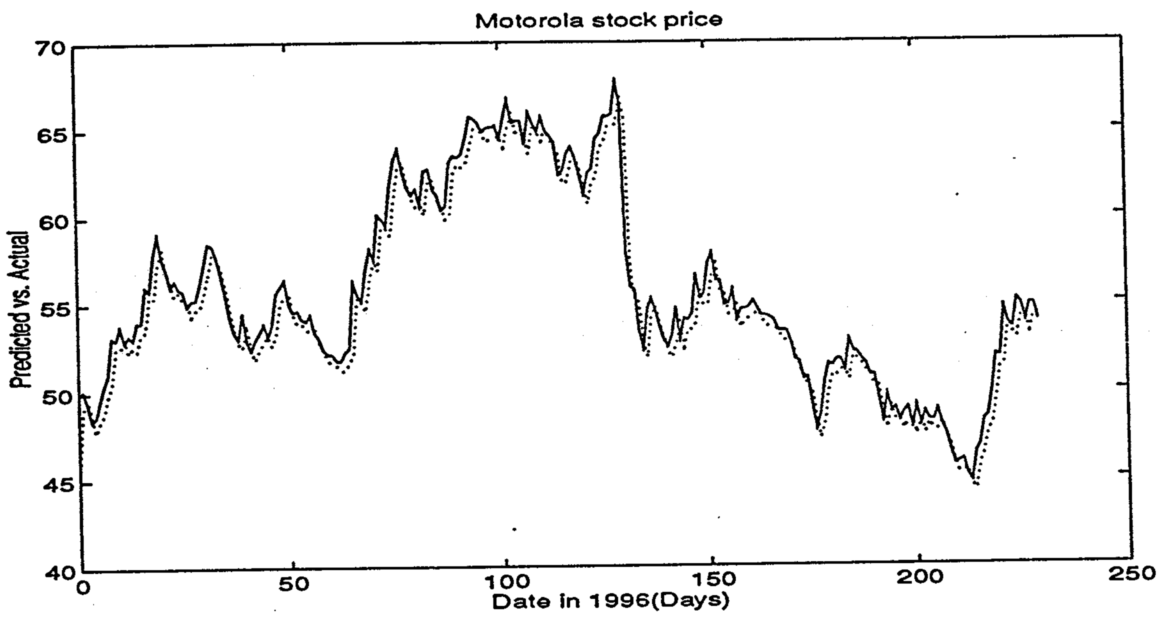


Fig. 9.— The predicted vs. actual stock prices of Motorola in 1996. The method is based on daily changes rather than prices. The solid line is the actual price; the dotted line is the predicted prices by the neural network.

A Neural Network Model of Jazz Improvisation

Todd J. Green
Yale University, Dept. of Computer Science
New Haven, CT, 06520

Abstract
Jazz improvisation is learned mostly by example. The target-note technique is often used as an explanation of this learning process. We outline here an implementation of this model of improvisation by means of a neural network. We use a modified version of Hopfield's network, based on the research conducted by Petri Toiviainen at the University of Jyväskylä, Finland but with an improved scheme for chord representation. We were unable to produce meaningful results with our network. Instead we include reproductions of Toiviainen's results.

1. INTRODUCTION

1.1. The Target-Note Technique

The target-note approach is a simple, commonly-used way of explaining the microstructure of an improvised jazz melody (Mehegan 1959; Toiviainen 1995). By "microstructure" we mean the structure of the melody taken one chord at a time, without regard for the solo's larger-scale structures (phrasing, dynamics, etc.).

The approach goes like this: to determine what notes to play in a given measure, the improviser picks a "target" note from the next chord (either the root or one of the upper tones) and applies a learned melodic pattern to play over the current measure, landing on the target note. To play over the next chord, the improviser starts with this note, picks a new target note from the next chord, and applies a pattern again; and so on.

1.2. Simplifying Assumptions

Several constraints can be imposed on the input without sacrificing too much of the model's generality. Firstly, we consider only compositions in common time (4/4). This is not too limiting since the great majority of jazz composed and performed in the bebop era is in 4/4 time (Mehegan 1962).

Second, we allow the half-note as the smallest harmonic unit, i.e., we assume there are no more than two chords per measure. This is also not too serious a constraint, as it is almost always the case in bebop music.

Thirdly, the smallest melodic unit we allow is the eighth note. The justification here is that in many cases shorter rhythmical units (16th, 32nd, etc.) can be viewed as simply ornamenting an underlying eight-note melody (Toiviainen 1995).

Finally, we allow only monophonic melodies (one note at a time). This too is not a severe restriction; the most significant solo instruments in jazz, the trumpet and the saxophone, are monophonic instruments.

2. ARCHITECTURE OF THE MODEL

2.1. A Modified Hopfield Network

One successful application of a neural network to implement this model used a modified Hopfield architecture. A 6-by-14 array of fully-interconnected neurons was used, where the six columns represent the note preceding the current half-measure, the 4 notes of the half-measure, and the note following the current half-measure, respectively. The neurons are self-coupled as well. The rows of the array represent the 12 tones, a rest, and a ligature (tie to previous note). When the network is stable, only one neuron per row is active.

Each neuron receives an additional, external activation from one of two other arrays, one representing the chord of the current half-measure and one representing the chord of the following half-measure (see 2.3).

2.2. Interconnections

The interconnections between neurons of the same column are inhibitory and fixed to ensure that only one neuron per column is active when the network has stabilized.

The interconnections amongst neurons of different columns are excitatory and dynamic; they are altered during the learning phase.

The interconnections from external chord neurons are also excitatory and dynamic, modified during the learning phase.

2.3. Chord Representation

2.3.1. Localized Representation

To represent chords in Toiviainen's architecture, two additional arrays of neurons are used, one for the chord of the current half-measure and one for the target chord. Each neuron in these arrays corresponds to exactly one chord type, e.g., maj7, 13#11, etc., and only one neuron per array is ever active. These chord matrices are fully connected to the neurons in the melody matrix.

2.3.2. Distributed Representation

In the bebop era, jazz musicians explored the use of chord substitutions to create added interest and dissonance to their compositions and improvisations (Lawn 1993). A soloist can spice up a solo by implying a substitute chord; i.e., playing notes that fit over a chord other than that specified by the lead sheet, but treating the chord as functionally equivalent to the original. The substitute harmony generally has at least two tones in common with the original chord; it works because it sounds like the original.

In order to increase the generality of the model, we decided to represent chords by their component notes. This way, chords that are similar (e.g., maj7 and maj6) have similar representations, allowing the network to handle simple chord substitutions.

More complicated but still often-used substitutions involve chords with a different root from the original. The most common example of this is the tritone substitution, which uses a dominant chord with the root a tritone from the original. The two chords sound alike because they have two notes in common; however, when they are transposed to the same root, their differences in harmonic function are obscured. In order to get around this limitation, we decided to represent all chords relative to the tonic. For instance, under the old scheme, Emin7 | A7 | Dmaj7, a ii-V-I progression in the key of D, would be fed into the network as Cmin7 (target F7) | C7 (target Fmaj7) | Cmaj7; under the new representation, the chords are fed in as Dmin7 (target G7) | G7 (target Cmaj7) | Cmaj7.

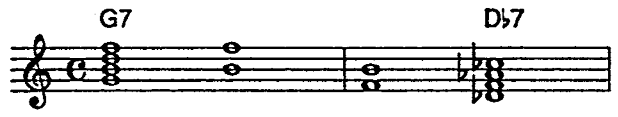


figure 2: example of a tritone substitution, showing common tones

2.4. Activation

The activation of a neuron j is defined by the functions:

```
a(j) = sgn(chord_a(j) + net_a(j))
```

where

```
chord_a(j) =   (activation of chord tone i) * (weight j, i)
             i

 net_a(j) =   a(i) * (weight j, i)
            i
```

2.5. Training

The learning phase is different in this implementation from a conventional Hopfield network. Learning does not take place all at once, but rather in epochs. To train on one half-measure, the appropriate entries within each chord activation matrix are set to maximum, and the activations for the notes to be associated with each chord are set to maximum.

The weights in the weight matrices are not computed all at once, as in the conventional Hopfield network (Haykin 1994). Instead, they are modified one at a time according to the following Hebbian rule:

```
w(j,i) += eta0 * a(j) * a(i)
w_external(j,i) += eta1 * a(j) * a(i)
```

where `w_external` is the weight between neuron j and chord neuron i.

2.6. Relaxation

The network is updated one half-measure at a time.

The neurons are updated asynchronously until the network stabilizes. At this point, one neuron in each column will be active, representing the note to be played on that beat. The last two notes of the network (the target notes) are fed back as the first two notes of the next half-measure, and the process is repeated.

4. EXPERIMENTS

We were unable to work all the bugs out of our network. Thus, we were unable to conduct any experiments. However, it is worth reporting the rather striking results of an earlier study (Toiviainen 1995). In this study, the researcher found that the limited storage capacity of his network made it impossible to train it on more than a few half-measures; too much training resulted in "cross-talk" between the exemplars. His solution was to use a cycle of 8 instances of the network, training the first on the first half-measure, the second on the second half-measure, and so forth. Since jazz solos tend to display phrasing on the scale of 4 bars (or 8 half-measures), this approach had the advantage of lending his results a thematic coherence that make them sound more authentic, even if attained by a fairly artificial means. Nevertheless, the network managed both to apply the patterns it was taught correctly (see boxed sections of figure 3c) and to generate new and stylistically-consistent patterns based upon those it was taught. The first two examples are from trumpeter Clifford Brown's solos on the standards "All the Things You Are" and "Gertrude's Bounce". The third solo is the output of the network, given the chord changes of the standard "I've Got Rhythm" as input.

We believe that our distributed, relative-to-tonic chord representation represents one of many refinements to be made in modeling the target-note technique with a neural network. It would be straightforward to extend the model to incorporate, for example, a representation of accent.

a

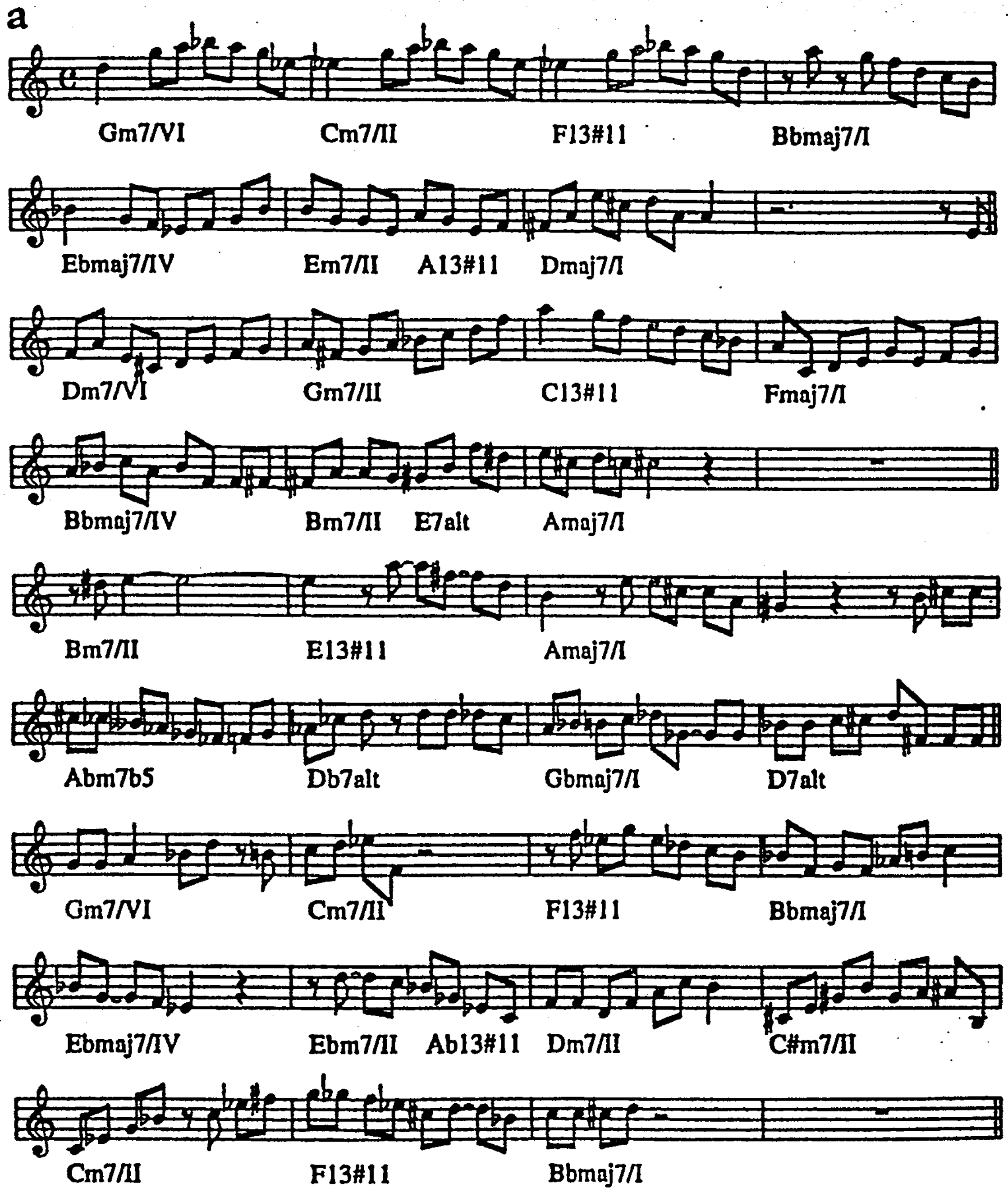


figure 3a: training set #1: excerpt from Clifford Brown's solo on "All the Things You Are"

b

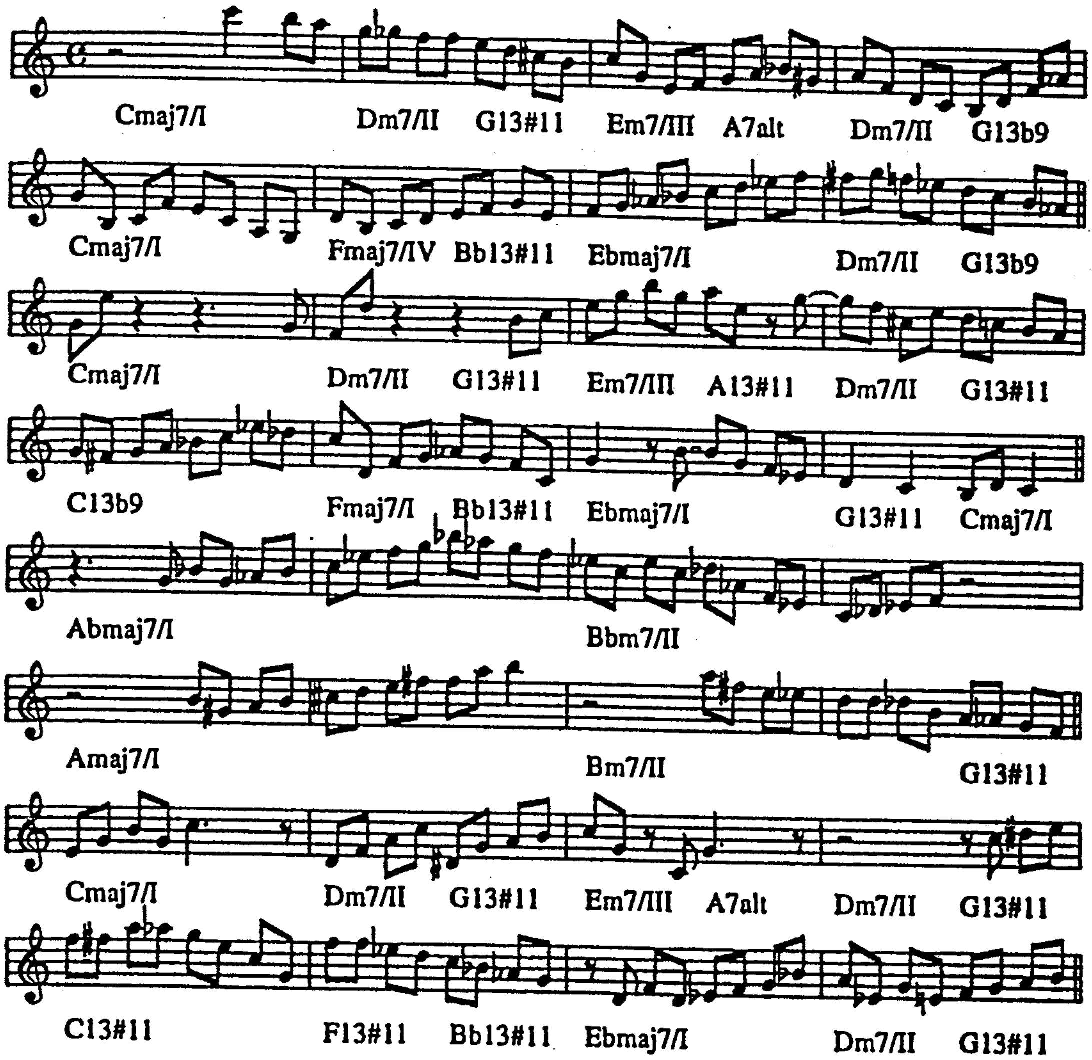


figure 3b: training set #2, from Clifford Brown's solo on "Gertrude's Bounce"

C

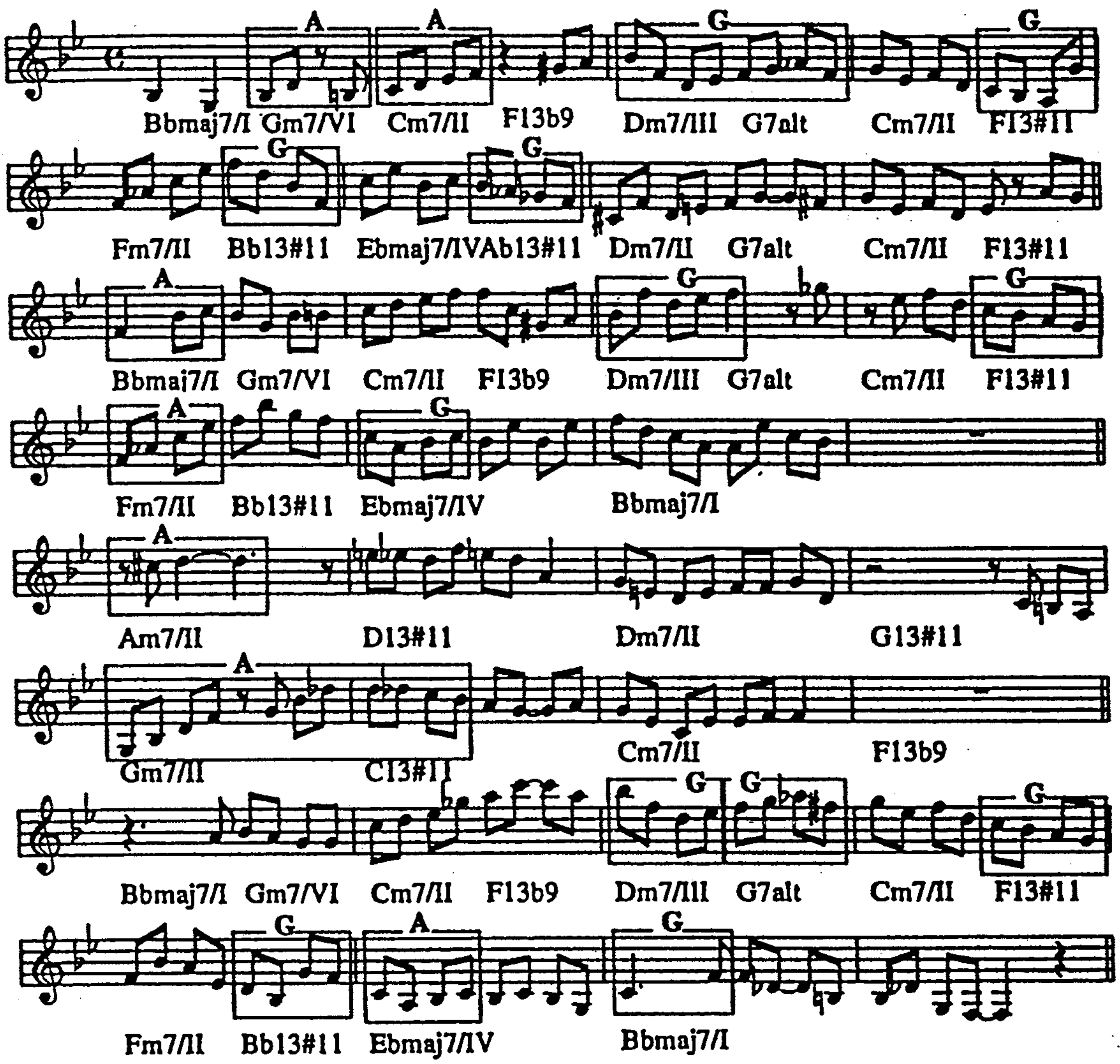

A
A
G
G
Bbmaj7/I Gm7/VI Cm7/II F13b9 Dm7/III G7alt Cm7/II F13#11
G
G
Fm7/II Bb13#11 Ebmaj7/IV Ab13#11 Dm7/II G7alt Cm7/II F13#11
A
G
G
Bbmaj7/I Gm7/VI Cm7/II F13b9 Dm7/III G7alt Cm7/II F13#11
A
G
Fm7/II Bb13#11 Ebmaj7/IV Bbmaj7/I
A
Am7/II D13#11 Dm7/II G13#11
A
Gm7/II C13#11 Cm7/II F13b9
G
G
G
Bbmaj7/I Gm7/VI Cm7/II F13b9 Dm7/III G7alt Cm7/II F13#11
G
A
G
Fm7/II Bb13#11 Ebmaj7/IV Bbmaj7/I

figure 3.c: output of the network (chord changes from "I've Got Rhythm"); rectangular boxes denote patterns quoted verbatim from training set ("G" indicates "Gertrude's Bounce" and "A" indicates "All the Things You Are").

ACKNOWLEDGMENTS

Eric James and Andro Hsu assisted in the recording of the examples.

REFERENCES

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation.* Englewood Cliffs, NJ: Macmillan Publishing Co.

Lawn, R. J., & Hellmer, J. L. (1993). *Jazz: Theory and Practice.* Belmont, CA: Wadsworth Publishing Co.

Mehegan, J. (1959). *Jazz Improvisation I: Tonal and Rhythmic Principles.* New York: Watson-Guptill Publications.

Mehegan, J. (1962). *Jazz Improvisation II: Jazz Rhythm and the Improvised Line.* New York: Watson-Guptill Publications.

Toiviainen, P. (1995). Modeling the Target-Note technique of Bebop-Style Jazz Improvisation: An Artificial Neural Network Approach. *Music Perception*, v. 12. pp. 399-413.

Character Recognition with BPN

Benyuan Liu
Yale University, Physics Department
New Haven, CT06511

Abstract

Artificial neural network can be trained with given exemplars to recall stochastic associations correctly. Among many algorithms, back-propagation net (BPN) is a general-purpose and commonly-used paradigm in neural networks. We describe application of the BPN to Chinese character recognition. The details of the input, output pattern, the BPN algorithm, and the net's performance are discussed.

1. Introduction

With the great increase of computer application in China, Chinese character recognition becomes a hot project,. Since the input of Chinese characters is very complicated and requires people to remember many rules. It's very inconvenient for people to write Chinese on computer sincethe speed is limited by tedious input methods. So, the implementation of Chinese character recognition will have a large effect on both people and the computer industry.

Unlike western characters, there are several thousand of Chinese characters, many of them being very complicated , and some of them may have very little difference in form. In this project I chosed ten simple characters [Fig1], used BPN to construct the neural net, and investigated the correctness of the net's recognition capacity.

2. Input Pattern Definition

The input pattern should represent the information of the characteristic of input object. In this case , the characters' structure and shape. It should be convenient to be entered into the computer. Perhaps the most commonly used method is to divide the character into a two-dimension video pixel matrix [Fig 2]. There are two possible states for each pixel, occupied or not. A logic "1" means the pixel is filled, logic "0" means it is empty. For character recognition, binary information is enough , we need not gray scale representation .

Due to the complexity of Chinese characters, we may need a large video pixel matrix to represent them , but in our application , we only choose a moderate resolution , say, 16 x 24. For a real product a high resolution will be necessary, because extra pixel information will give the neural net more information that can be used to distinguish different characters.

We collected the hand-writing of four people for each of the ten characters. A scanner is used to scan the characters in as a black-and white picture. Then we use a graph software(xv) to pixelize each character and scale the size to a 16x24 matrix.

Then we concatenate the 24 row vectors into a one dimensional 384(16x24) elements vector [Fig 2], which can be input into computer easily.

3 Output Pattern Definition

Since our exemplar only consists of 10 characters, the most straightforward solution is to set up a one - to - one mapping from input space to output space [Fig 3] . Then we only need ten output neurons. Given one input pattern, one, and only one output neuron will become active, the others remaining inactive. The position of the active neuron specifies the input character. The advantage of this approach is obvious, as stated in *Building Neural Networks* [refrence 1], "It is relatively easy to construct a layer of processing elements in a neural network that will produce a one-and-only-one output across the layer. It is also very easy for the external application to interpret the output from the network: It need only determine which output unit is active, and then use the character associated with the active unit. "

There is an obvious disadvantage in this approach. As we have mentioned, there are thousands of Chinese characters. If we use the one-to-one mapping in a real product , there would be several thousand output units in the net. That means we need a huge weight matrix and the training will take an extremely long time. More importantly, if we want to add a new character to input space, the output space must be extended to contain a new unit corresponding to the newly added character, and the training must be repeated once again. In fact , there are several kinds of code methods for Chinese characters. For a real application , it's very important to select a proper code for the input character. For different choice, the performance can be very different. We won't discuss this point any further, we will focus on the simple application of ten characters, in which it makes sense to choose ten one-to-one output units.

4. Back-Propagation Algorithm

In BPN, each hidden and output neuron is designed to perform the following functions.

1. Compute the internal activation and the output of the neuron. This is the forward direction.
2. Compute the difference between the real and desired output, use it to adjust the relative weights. This is the backward direction.

The algorithm can be described in following data flow chart [Fig 4].

5. Network Specification

We use three layers for this application. First, the input layer , has 384 units. It contains the characters' pattern (shape) information, and acts as a fan out layer. We have only one hidden layer. It is basically a feature identification layer. The third is a ten unit output layer. It's just a binary indicator of the input characters. After the proper training, we expect that for each input, a corresponding output unit will fire, the others remaining inactive. Each of the input nodes is connected to all the hidden nodes, and each of the hidden nodes is connected to all the output nodes.

Since the input and output patterns are basically binary signals, we employ a sigmoidal non-linear function as the activation function for both hidden and output layers. We use the non-linearity of the sigmoidal function in hidden layer to train the net to be able to differentiate non-linear input patterns.

We set the neuron number in the hidden layer to be a fraction of the number of the input neurons. Then , according to the quality of the performance , we adjust the number up or down.

6. Network Performance

Use the four sets of exemplars [Fig 1] to train the net. After 500 training epochs, we investigate the net's ability to recall the training character patterns correctly[table 1], Ideally, given a specific input pattern , the output should have one active unit, the others should all be inactive. From the following table, we can see there is only one large output whose value is great than 0.9, the others' value is at the order of 0.01[Fig5]. The result is very impressive.

We use another test character [Fig1] set not included in the training set, and list the results in the following[table 2]. We can see from the table that all ten characters are recognized correctly. However, the performance is not as good as for the training set. The amplitude of the active neurons becomes smaller, but still remains the dominating one[Fig 6]. At the same time, the amplitude of the some inactive neurons get larger , especially those units corresponding to a similar character. For example, the character 1 is similar to character 5 in structure, in the output pattern of character 5, we find a large increase of neuron 1, which is corresponding to character 1[Fig 7].

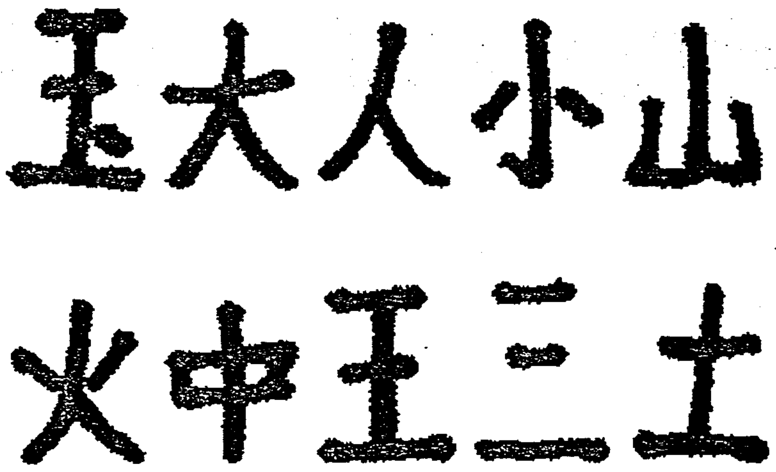
In order to get a high ratio of recognition in real application, one may need to collect enough sets of handwriting, so the net can be trained to store more information that can be used to identify many different features of a character.

This project is for course CS477. Thanks for the help and encouragement from Professor Miranker. My thanks also to TA Hong Xiao for the helpful discussions. Lastly thanks to those people who generously provide their hand-writings.

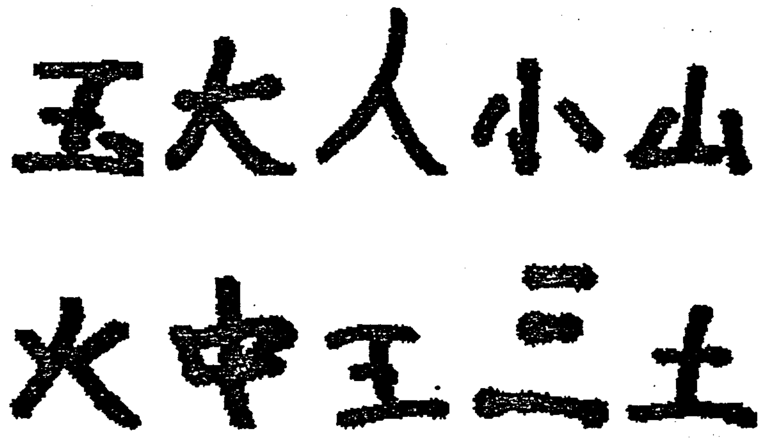
References

1. David M. Skapura , *Building Neural Networks.* ACM Press 1996
2. Simon Haykin , *Neural Networks----a Comprehensive Foundation.* Maxewell Macmillan Canada 1994
3. Dennis W.Ruck, *Science of Artificial Neural Networks II.* 1993 Orlando

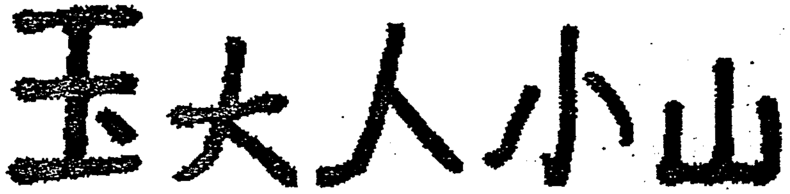
Training exemplar set 1:

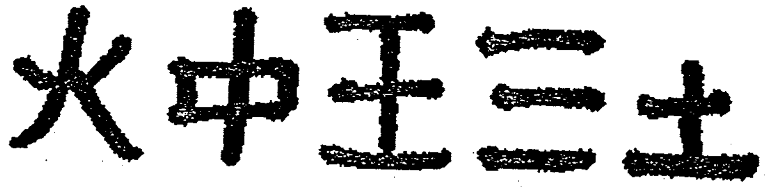


Training exemplar set 2:

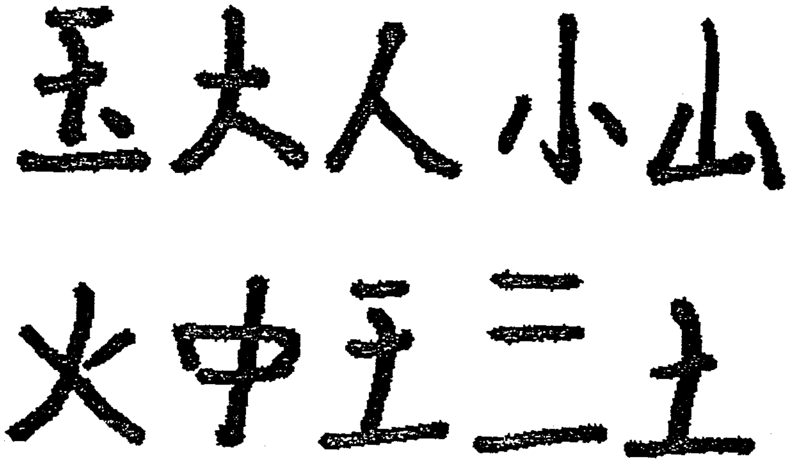


Training exemplar set 3:





Training exemplar set 4:



Testing character set

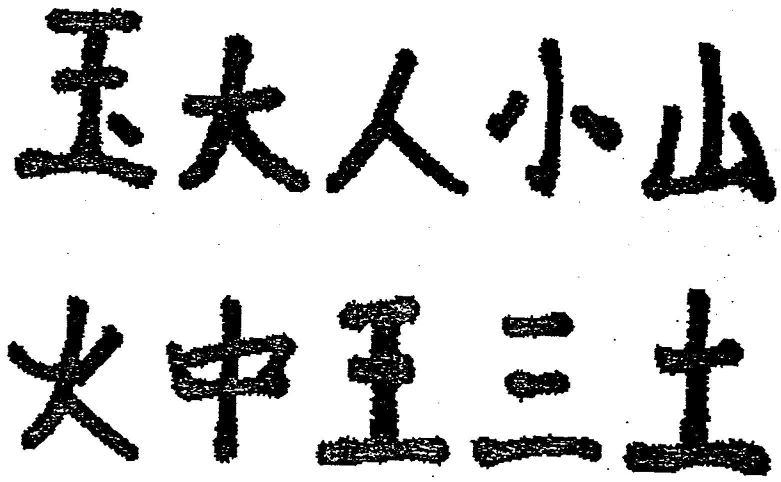


Fig 1. Training and testing characters.

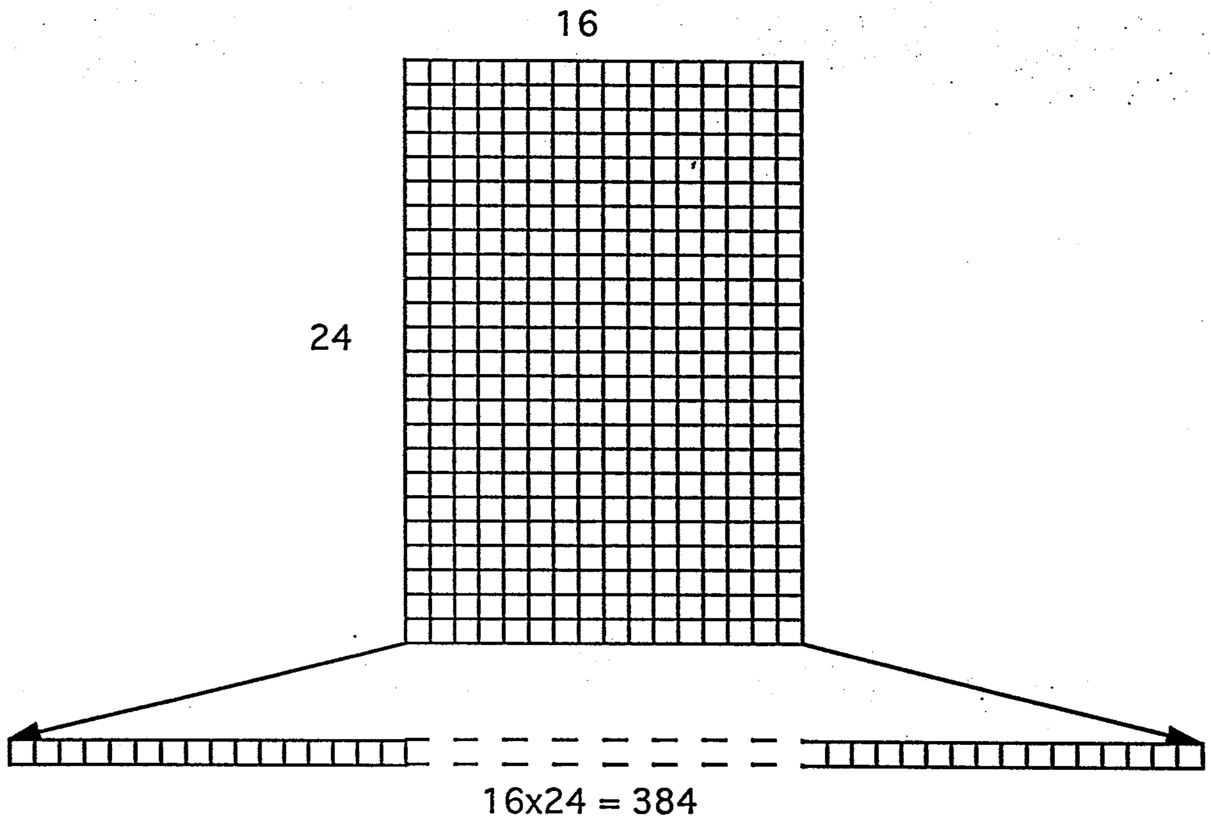


Fig 2 This diagram illustrate the two dimension video pixel matrix of a character(16x24). And how we concatenate the matrix rows to form a one dimensional array.

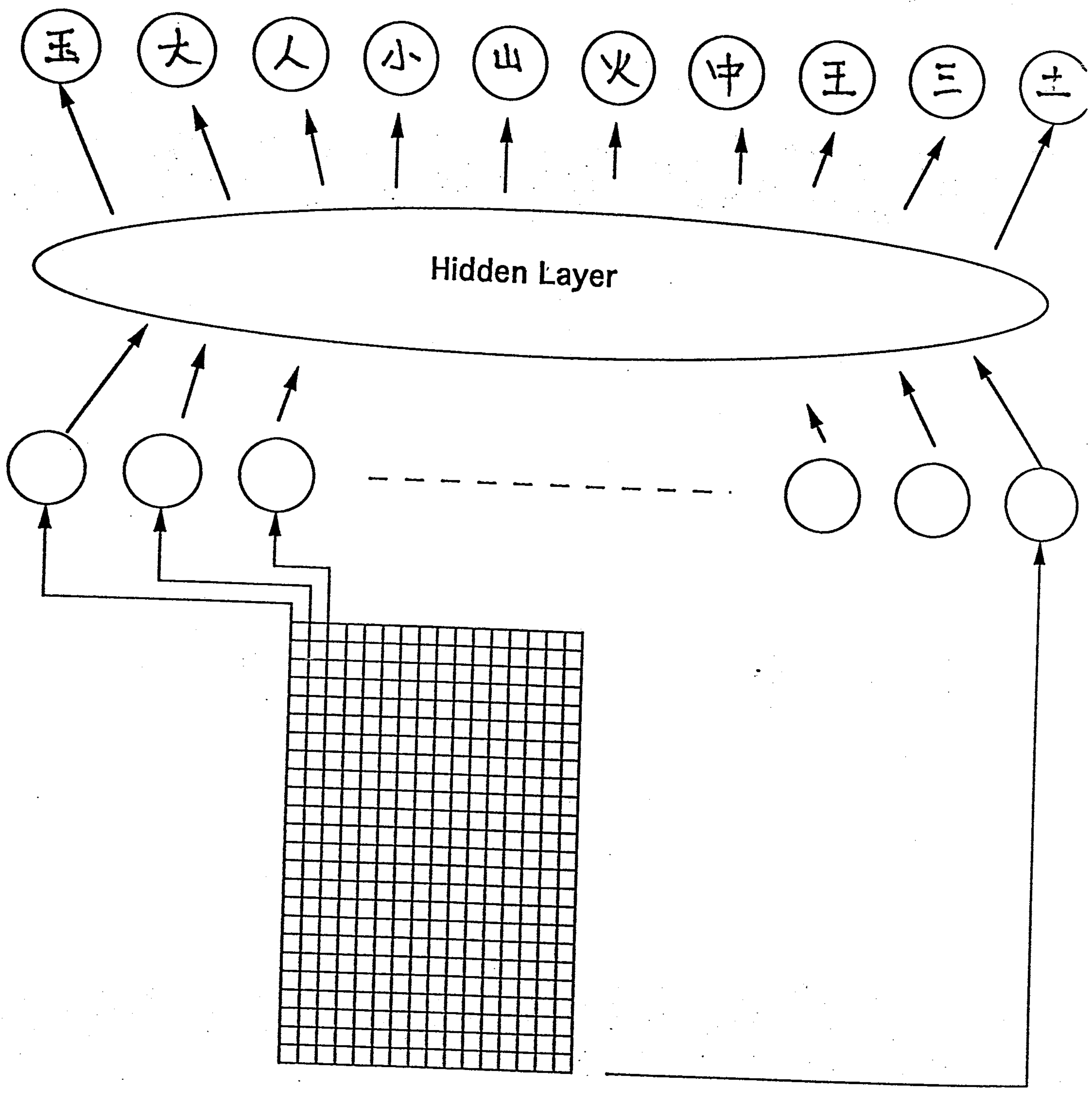


Fig3. This diagram shows the output classification scheme. There are ten output units, each one corresponds to one input character, after the net has been trained properly,. only the unit which is associated with the input character will be fired

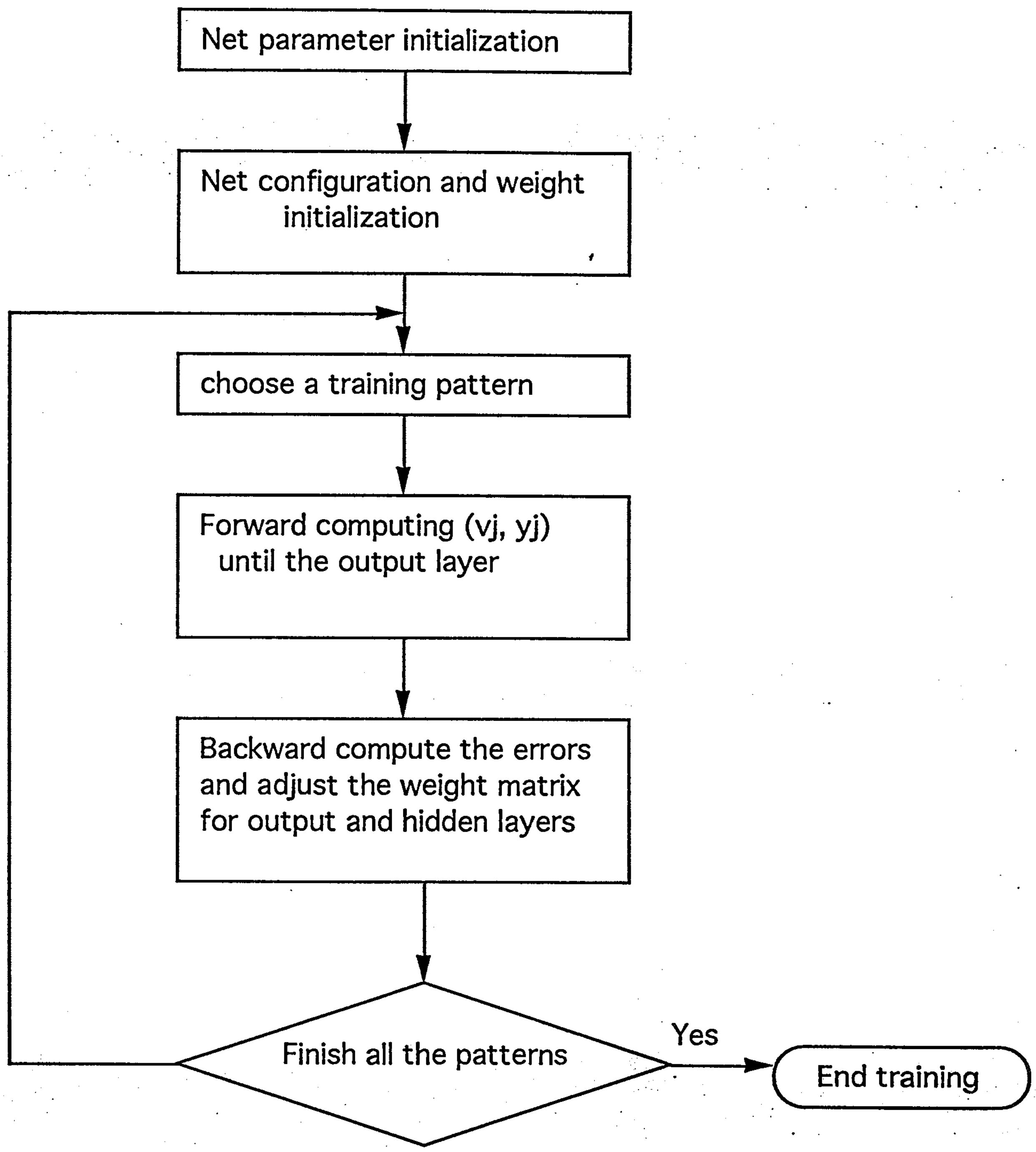

Net parameter initialization
Net configuration and weight initialization
choose a training pattern
Forward computing (vj, yj) until the output layer
Backward compute the errors and adjust the weight matrix for output and hidden layers
Finish all the patterns
Yes
End training

Fig 4. Data flow chart of the back-propagation learning algorithm.

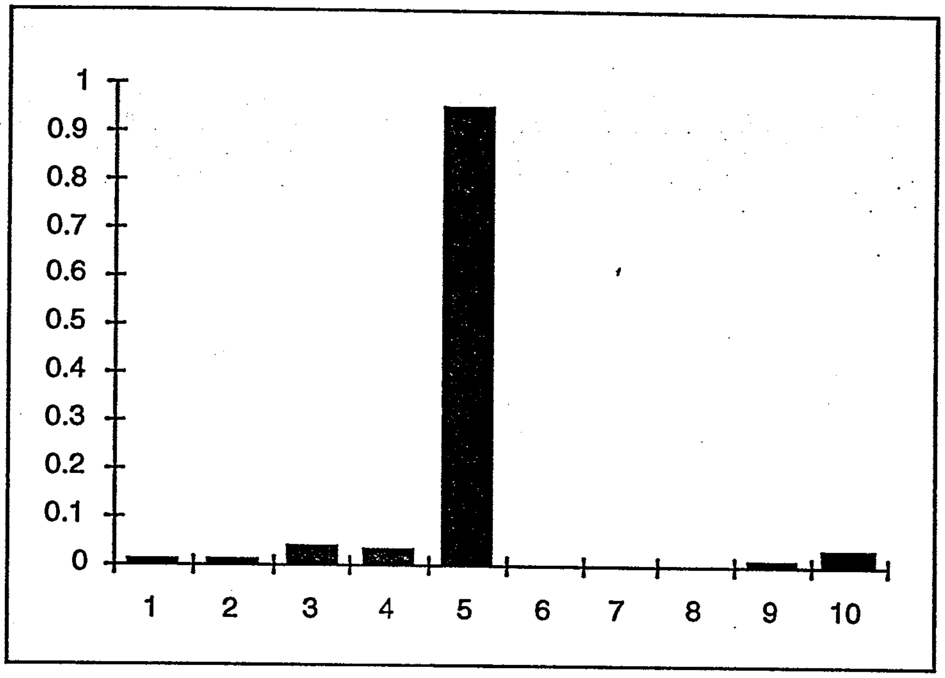


Fig 5. Chart of the output units of Exemplar 35(the fifth character), we can see the fifth neuron of the output layer is active(near 1), the others is very small.

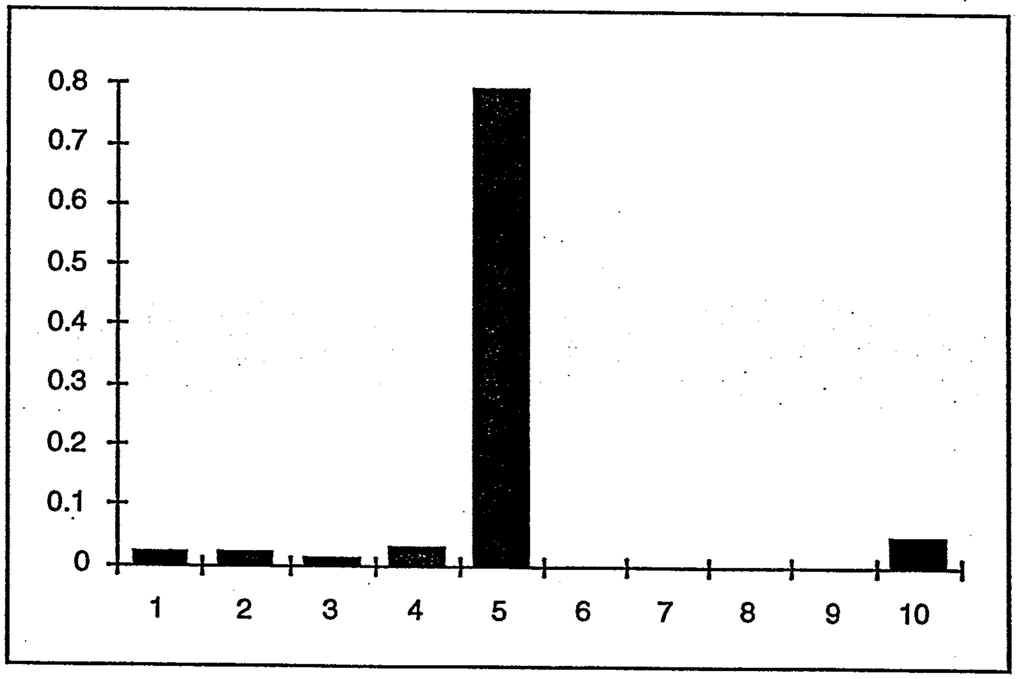


Fig 6. Output neurons of test5, the amplitude of the active unit becomes smaller, but still remain dominating.

Output	0	1	2	3	4	5	6	7	8	9
Ex 1	0.96	0.02	0.01	0.00	0.01	0.00	0.00	0.02	0.02	0.00
2	0.03	0.94	0.02	0.00	0.00	0.05	0.01	0.00	0.00	0.02
3	0.00	0.03	0.95	0.00	0.01	0.03	0.00	0.01	0.00	0.00
4	0.00	0.00	0.00	0.95	0.01	0.00	0.05	0.05	0.00	0.02
5	0.00	0.00	0.03	0.02	0.96	0.00	0.00	0.00	0.03	0.01
6	0.00	0.05	0.01	0.00	0.00	0.96	0.04	0.00	0.00	0.03
7	0.00	0.01	0.00	0.04	0.00	0.03	0.96	0.00	0.00	0.01
8	0.04	0.00	0.00	0.03	0.00	0.00	0.00	0.94	0.03	0.03
9	0.05	0.00	0.02	0.00	0.02	0.00	0.00	0.03	0.95	0.02
10	0.01	0.01	0.00	0.00	0.02	0.00	0.00	0.02	0.03	0.95
11	0.96	0.01	0.01	0.00	0.02	0.00	0.00	0.03	0.03	0.01
12	0.01	0.94	0.02	0.00	0.01	0.04	0.01	0.00	0.00	0.03
13	0.01	0.03	0.96	0.00	0.02	0.02	0.00	0.00	0.00	0.00
14	0.00	0.00	0.00	0.95	0.01	0.00	0.04	0.02	0.00	0.01
15	0.01	0.04	0.03	0.04	0.94	0.00	0.00	0.00	0.00	0.02
16	0.00	0.02	0.02	0.00	0.00	0.96	0.03	0.00	0.00	0.01
17	0.00	0.03	0.00	0.02	0.00	0.03	0.97	0.00	0.00	0.01
18	0.02	0.00	0.01	0.03	0.01	0.00	0.00	0.95	0.01	0.02
19	0.01	0.00	0.01	0.00	0.02	0.00	0.00	0.01	0.96	0.04
20	0.01	0.02	0.00	0.00	0.03	0.00	0.00	0.03	0.02	0.94
21	0.95	0.02	0.00	0.00	0.01	0.00	0.00	0.04	0.01	0.01
22	0.02	0.93	0.03	0.00	0.01	0.04	0.01	0.00	0.00	0.01
23	0.02	0.03	0.94	0.00	0.03	0.02	0.00	0.00	0.01	0.00
24	0.00	0.00	0.00	0.93	0.05	0.00	0.04	0.02	0.00	0.01
25	0.00	0.00	0.01	0.02	0.95	0.00	0.00	0.00	0.02	0.02
26	0.00	0.03	0.04	0.00	0.00	0.95	0.02	0.00	0.00	0.00
27	0.00	0.03	0.00	0.05	0.00	0.01	0.95	0.00	0.00	0.01
28	0.04	0.00	0.00	0.04	0.00	0.00	0.00	0.93	0.03	0.01
29	0.01	0.00	0.02	0.00	0.01	0.00	0.00	0.05	0.96	0.00
30	0.00	0.01	0.00	0.00	0.01	0.01	0.00	0.01	0.01	0.95
31	0.93	0.03	0.01	0.00	0.00	0.00	0.00	0.04	0.02	0.01
32	0.04	0.95	0.04	0.00	0.01	0.03	0.00	0.00	0.00	0.03
33	0.00	0.03	0.94	0.00	0.04	0.02	0.00	0.01	0.02	0.00
34	0.00	0.00	0.00	0.96	0.03	0.00	0.02	0.04	0.00	0.01
35	0.01	0.01	0.04	0.03	0.95	0.00	0.00	0.00	0.01	0.03
36	0.00	0.05	0.04	0.00	0.00	0.94	0.01	0.00	0.00	0.02
37	0.00	0.01	0.00	0.02	0.00	0.02	0.96	0.00	0.00	0.01
38	0.03	0.00	0.01	0.02	0.00	0.00	0.00	0.92	0.04	0.07
39	0.01	0.00	0.01	0.00	0.01	0.00	0.00	0.04	0.96	0.02
40	0.02	0.03	0.00	0.00	0.02	0.00	0.00	0.05	0.01	0.92

Table 1. The output of the 40 training patterns, we can see the result is very good. The active unit is over 0.90, and the inactive unit is at the order of 0.01. The contrast is quite obvious, the application can easy tell what's the input character by the output indicator.

Output	0	1	2	3	4	5	6	7	8	9
test 1	0.69	0.01	0.01	0.00	0.01	0.00	0.00	0.04	0.07	0.05
2	0.01	0.87	0.05	0.00	0.00	0.07	0.00	0.00	0.00	0.02
3	0.00	0.10	0.73	0.00	0.01	0.13	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.85	0.01	0.00	0.05	0.06	0.00	0.03
5	0.02	0.02	0.01	0.03	0.79	0.00	0.00	0.00	0.00	0.05
6	0.00	0.26	0.01	0.00	0.00	0.72	0.03	0.00	0.00	0.04
7	0.00	0.01	0.00	0.03	0.00	0.08	0.88	0.00	0.00	0.01
8	0.03	0.00	0.00	0.01	0.00	0.00	0.00	0.58	0.07	0.15
9	0.05	0.00	0.03	0.00	0.03	0.00	0.00	0.09	0.76	0.02
10	0.01	0.01	0.00	0.01	0.01	0.00	0.00	0.20	0.03	0.81

Table2. The output units of the test character set,. The amplitude of the active neuron becomes smaller, but inactive neurons remain at the order of 0.01.

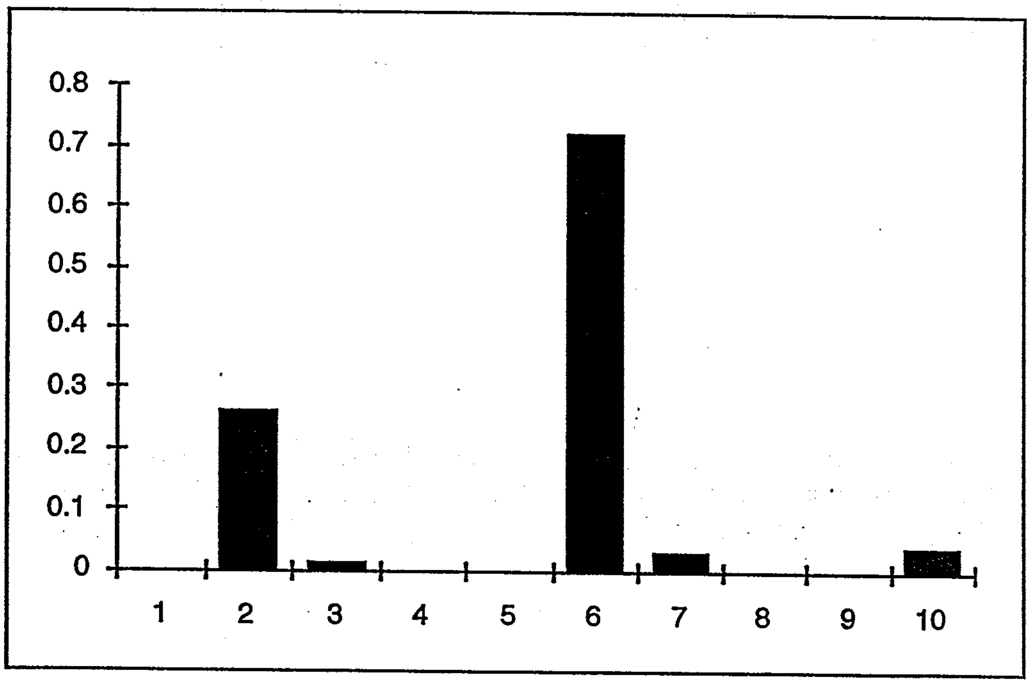


Fig7. Output of character 5, which is similar to character 1 in feature. We see the output unit corresponding to character1 get larger.

Hybrid Networks

Josh Mahowald
Yale University, Computer Science 477-Neural Networks
P.O.Box 202545 New Haven, CT 06520

Abstract

Two powerful, commonly used neural nets, the Hopfield network, and the bi-directional associative memory (BAM) are excellent models to study theories of memory. Hopfield networks clear up "fuzzy" memories, while associative memory networks relate one "picture" to another. This project attempts to combine both Hopfield networks with BAM networks to determine whether the two networks can be mutually beneficial, and expand each others capabilities.

Introduction

There were actually two goals for this project. The first was to create an environment which was conducive to testing both Hopfield and BAM networks, allowing the user to manipulate the data sets used at ease, as well as to observe graphical outputs of both networks. The second goal was to conduct preliminary experiments that investigated the possible benefits of using BAM and Hopfield networks in conjunction with one another.

The Hopfield network is an excellent tool for clearing up "noisy" patterns, while the BAM is good at remembering relationships between patterns. The BAM, however, does not have within it a method to correct errors without supervision. Therefore, by using a Hopfield network as a preprocessor and/or postprocessor for a BAM network, we can improve the BAM network's ability to correctly retrieve patterns given a "noisy" input.

It is also possible that a BAM network can increase the performance of a Hopfield network. If the BAM network remembers a relationship between x and y, and the network is given x' such that x' is closer to the pattern x than any other input vector pattern, the network will produce a vector y' that is closer to y than any of the other output memories the BAM network was trained on. If, by sending the output vector y' back through the BAM network to the Hopfield network, some of the noise was to be cleared up, we can exploit the fact that the BAM network can use massive parallel processing for its computation, updating all neurons in a constant number of steps, while the Hopfield network, which relies on asynchronous updating, requires time that is proportional to the number of neurons in the system.

Implementation

The data vectors used in this project were 120-dimensional. Users create an input pattern by drawing on a 10 x 12 pixel palette. Output patterns are also displayed on the same palette. Each pixel represents a neuron, where a value of +1 indicates a filled in pixel, and -1 indicating a blank pixel.

There are actually two user-interface based programs[1] that were created for this project.. One, called HopTrain, is used to set the fundamental memories of the two

[1] The code for the palette that allows the user to draw on it was graciously donated by Kishnan Nedungadi.
Both programs can be retrieved from
http://powered.cs.yale.edu:8000/~mahowald/HopBam/Project.html

Hopfield networks, and to create the BAM network that creates mappings between the pairs of fundamental memories for the Hopfield networks. The other, called HopBam, tests the three networks, and allows the user to view the fundamental memories and to randomly distort them.

To create the Hopfield networks used, the algorithm in Haykin (8.9) was used. Asynchronous updating was used to guarantee convergence of the network to stable states. For each iteration, the network picks a neuron at random, and updates its state according to the algorithm outlined in Hayken, 8.4-8.6. This is repeated for all the neurons. Once all neurons have fired, the network checks to see if the state of the network has changed. If so, the iterations continue, if not, they stop.

The BAM network is based on the associative memory network described in Haykin 3.3. The memory matrix is calculated as the sum of the outer products of the correlated memories. Recall is then simply the application of a memory state to the memory matrix. Because of the graphical nature of our example, there is hardlimiting that is applied during recall. Specifically, since our neurons are only in the state of being on or off, and never any gradients, the output state of the BAM is correspondingly transformed.

It should be noted that in both cases, because of the graphical nature of the application, the network state is not a vector of neurons, but rather a matrix of neurons. This therefore makes the corresponding weight matrices into 4 dimensional arrays.

Experiments

The results experiments that were conducted were positive, indicating that there were benefits to be gained from using the two types of networks in conjunction with one another. The use of both post and pre-processor Hopfield networks on the BAM network greatly increased the production of correct output patterns for the data-set that was used. Also using the BAM network as a preprocessor to a Hopfield network showed a possibility for increasing the efficiency of a Hopfield network. The data set consisted of eight pairs of fundamental memories, eight fundamental memories taken from Haykin 8.4 and the other eight generated by the author[2].

Before continuing with a description of the experiments run, a couple of clarifying are in order. Patterns will be referred to by the index of the pair that they are in (1-8), and by either L or R, to denote a fundamental memory of the left Hopfield network or the right Hopfield network. The measurement of error for the BAM network will be given by the number of neurons in incorrect states compared to the desired output. For the Hopfield network, we have four, more subjective measurements. For these measurements we define two patterns to be "close" if they differ by no more than 12 pixels, 10%. Error type one is when the Hopfield network ends up in a spurious state that is close to the desired output. Error type two is when the Hopfield network ends up in a different pattern than the desired one, but that pattern is still a legitimate fundamental memory. Error type three occurs when the output pattern is close to a legitimate fundamental memory that is not the desire one. The fourth type of error is when the Hopfield network ends up as a spurious state that is not close to one of the fundamental memories.

The first experiment tested the ability of the BAM network to correctly retrieve an output pattern given a clean input pattern. In 10 of the 16 possible mappings the BAM

[2] The patterns can be seen by a browser at the URL:
http://powered.cs.yale.edu:8000/~mahowald/HopBam/Patterns.html

performed perfectly. In the 6 remaining mappings, all but one of the incorrect patterns were corrected once put through a Hopfield network as a postprocessor. The number of incorrect neuron states for the remaining 6 mappings is below. The arrows below indicate the direction of the mapping from the input to the output pattern

Table 1

	1L<-1R	3L->3R	4L->4R	5L->5R*	6L->6R	8L->8R
#Incorrect	5	9	6	5	6	3

The starred mapping indicates that the Hopfield network there reached a spurious state that was 2 pixels off the desired state—still better than the BAM alone.

The next experiment tested the Hopfield networks ability to correct noisy patterns, to determine its use as a preprocessor for a BAM network. For each of the 16 patterns 10 trials of noisy patterns were run[3]. The fundamental memories were distorted by changing each pixel in the pattern with a 25% probability. The numbers of occurrences of each type of error was recorded, in addition to the average number of neurons that fired until a stable state was reached. For errors of type 1, the average number of incorrect neurons is given after the total number of errors. For errors of type 2, the produced fundamental memory is listed.

[3] The results for the individual experiments can be seen at http://powered.cs.yale.edu:8000/~mahowald/HopBam/Ex2Results.html

Table 2

	1L	1R	2L	2R	3L	3R	4L	4R	5L	5R	6L	6R	7L	7R	8L	8R
E1							1,10	1,5		5,2		2,9	4,6		3,6	
E2			1,6		1,6											
E3																
E4	1		1	1		2										

The Hopfield networks achieved the desired state, or a close spurious state in 153 out of 160 trials, for a respectable 95%. In all of these cases the output of the BAM network, being deterministic, was identical to the output of the BAM network given perfectly clean input patterns can be found above in Table 1. All other cases used a BAM network with a postprocessor Hopfield network to produce the final output. In the five cases when a spurious state that did not resemble one of the desired output patterns the pattern after processing by the Hopfield network was not close to the desired output. In the two cases that resulted in a legitimate, but different fundamental memory than what was expected, the BAM network succeeded in exactly replicating the produced patterns corresponding image. With the exception of 5R patterns and one other exception, the resultant patterns of Type 1 errors were the desired output once passed through the BAM network and a postprocessor. Using a BAM network and Hopfield postprocessor on preprocessed patterns of 5R that resulted in a Type 1 error caused the final state to be 3 pixels off of the desired state. There was one case in which a failure to correctly preprocess a pattern of type 7L sent into a BAM network followed by a Hopfield postprocessor resulted in a spurious state that was not close to the desired final state.

The next experiment removes the preprocessor, and compares the ability of the BAM network to resolve noisy input patterns on its own, and when it is coupled with a postprocessor Hopfield network. Once again 10 trials were ran for each mapping, with the noise being produced by randomizing a fundamental memory pattern with a 25% probability[4]. In approximately 25% of the trials the BAM network successfully cleared up noisy patterns to produce the correct output pattern. In 11.25% of the trials the Hopfield postprocessor either failed to improve on the performance of the BAM network or made it worse. In all but one of those cases, however, over 10% of the neurons were in the wrong state after the noisy pattern was transformed by the BAM network. In approximately 4% of the trials the processing by the Hopfield network improved on the performance of the BAM network without producing the desired output pattern exactly. And in the remaining 35% of the cases, the Hopfield postprocessor cleared up all the noise that the BAM network failed to clear.

The next experiment compared the efficiency of using solely a Hopfield network to retrieve a memory to using a Hopfield network in conjunction with a BAM network. Because the BAM network can do all of its processing in parallel the time it takes to compute the output pattern given an input pattern is constant with the number of neurons. A Hopfield network, however, needs time proportional to the number of neurons because of its asynchronous updating. By feeding a noisy pattern into a BAM, and then feeding the BAM's response back into itself and giving that output to the Hopfield network, the hope is that the number of neurons fired by the Hopfield network to correctly retrieve the

[4] The results for the individual experiments can be seen at
http://powered.cs.yale.edu:8000/~mahowald/HopBam/Ex3Results.html

pattern decreases without increasing the number of incorrect neuron states. Once again 10 trials were used, with noise in a pattern being created at a rate of 25%[5]. In the table below #f1 represents the average number of iterations required by the Hopfield network to correctly retrieve the desired output, and #f2 the average number required when a BAM network is used as a preprocessor to the Hopfield network. The experiment also found that there was a decrease in errors when the BAM network preprocessed signals for the Hopfield network.

Table 3

	1L	1R	2L	2R	3L	3R	4L	4R	5L	5R	6L	6R	7L	7R	8L	8R
#f1	255	235	129	156	134	283	144	175	143	126	137	164	194	154	120	223
#f2	98	110	15	29	42	218	56	99	75	89	62	134	65	60	4	183

Conclusion and Analysis of Results

The ability of a BAM network to perfectly retrieve an output pattern from a noisy input pattern jumps from 25% by itself, to 60% with a postprocessor Hopfield network, and 95% with both a pre and postprocessor Hopfield network with the data set examined. This large jump is indicative of positive gains that may be gained by using Hopfield networks in conjunction with BAM networks, and could be applied to any area in which BAM's are currently used such as image databases. There also was positive evidence in

[5] The results for the individual experiments can be seen at http://powered.cs.yale.edu:8000/~mahowald/HopBam/Ex4Results.html

the use of BAM's to process noise as well, at least for small, relatively orthogonal data sets.

A caveat empor is due at this time. The data set examined was relatively small, and picked to get maximum results out of both networks. The set of patterns used was highly orthonormal, and similar results should not be expected from sets that are non-orthonormal. As mentioned in the introduction, however, one of the goals of this project was to create an environment that would easily allow experiments, and the author believes that he has been at least partially successful in this regard. As mentioned in the footnotes, the programs used as well as the data sets can be retrieved from and instructions for usage can be found at:

http://powered.cs.yale.edu:8000/~mahowald/HopBam/project.html

Hopefully, with a minimal amount of time, users can easily create their own data sets to refute or support the findings of this paper.

One final note, this project would not have been successful without the generous help of Kishnan Nedungadi, who made the graphical user interface part of the program possible.

—

Character Recognition Using Neural Networks

Kishnan Nedungadi
Yale University
Dept. Of Computer Science

1. INTRODUCTION

With the growth in size and speed of the Internet, almost any information can be found on the World Wide Web. The importance of digitally storing information has tremendously increased in the recent years. However there are several documents that only exist on paper and need to be transformed into digital medium. Retyping this information would be extremely tedious, expensive and probably impossible, and this is where character recognition comes into play. Today there are several character recognition application that solve just this problem.

This paper describes a character recognizer application that I created using Neural Networks. This application recognizes a subset of characters in the English Alphabet. Specifically, I have trained the network to recognize all the capital letters from A to Z. The primary purpose of this application is to understand how Neural Networks can be used to recognize characters. This application can easily be extended to include more letters, and numbers. Inorder to make this application a more practical and useful tool, I created a graphical user interface that allows the user to graphically input characters, and view the results.

2. INPUT PATTERN DEFINITION

Among the first problems I tackled was how the user should define an input pattern. Several character recognizers have successfully managed to capture the input in the form of a bitmap of 0's and 1's, and I have followed the same path in this application. The input pattern in this character recognizer application is represented by a series of bits. Specifically, there are 143 bits in the input pattern. These bits represent a 11 x 13 grid in which the input characters are represented. For instance, the letter 'I' could be inputted by the following bit pattern.

```
0 0 1 1 1 1 1 1 1 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0
0 0 1 1 1 1 1 1 1 0 0
```

$\xrightarrow{\text{Represents}}$

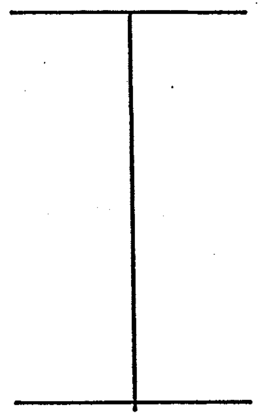


Figure 1: Input character Definition

I found this resolution of 11 by 13 to be sufficient to represent the characters that I wanted to train the network with.

3. OUTPUT PATTERN DEFINITION

Now that the input pattern definition is defined, I needed a method to define what the output of the network should be, and how it should be interpreted. The output should of course be a representation of the letter that is recognized. Since there are 26 characters that are recognized by the application, there should be 26 unique output representations. I have represented the output pattern using the *classification* scheme in which one, and only one output element is active for any given input pattern. The position of the active output element will determine the corresponding character. The output pattern is represented by a series of 26 bits each representing a unique character. An example of an output pattern is as follows:

The letter 'I' is represented in the output pattern by:

```
0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
              ↑
          Bit representing I
```

Figure 2: Output pattern Definition

4. NETWORK SPECIFICATION

In selecting the type of network that I should use for this application, I realized that I should use network that can recognize the features of the input that tend to correctly classify image patterns. This pattern matching characteristic must be an essential part of the type of neural network that is chosen. Literature tells us that the *Back Propagation Network* does an excellent job of doing just this. This is why I have selected the BPN as the type of network to use in this application.

Given the Input and the Output pattern definitions, my neural network will have 143 nodes as inputs, and 26 nodes as outputs. I estimated that about 1/3 number of hidden neurons should be sufficient to map the input characters the their corresponding output values. Therefore I have used 50 hidden neurons in my Character Recognizer.

The structure of the Back Propagation Network that I have used in the Character Recognizer is shown below:

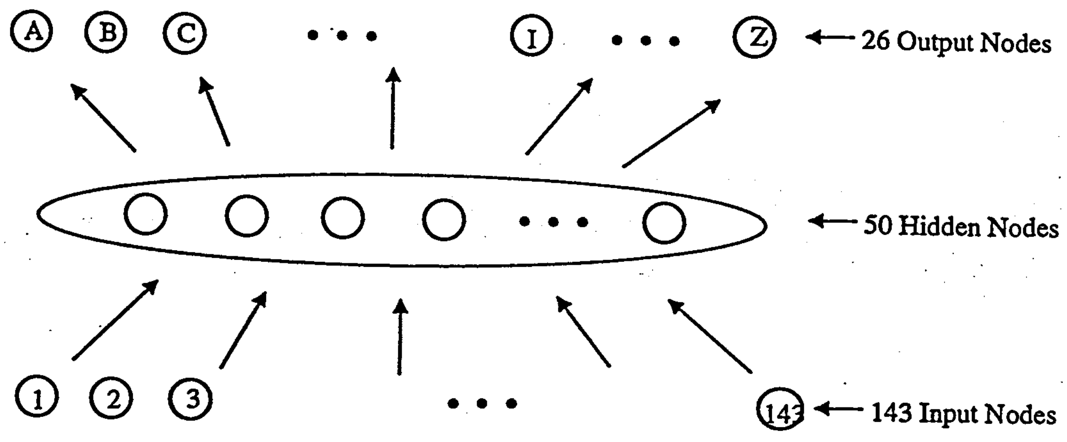


Figure 3: Structure of Neural Network

I have not represented all the connections between nodes so that the figure remains clean. In reality, every input node is connected to every hidden node, and every hidden node is connected to every output node.

Since the output units in the above network are binary, I used a sigmoidal activation function on the hidden, and the output layers. The sigmoidal function that I used in the network is:

$$f(net_i) = \frac{1}{1 + e^{-(neti)}}$$

5.TRAINING

My training data consisted of characters that I generated myself. I started the learning process by training the application only one of each of the characters, and gradually increased the number of exemplars based on what characters weren't recognized accurately. In my final training data set, I had generated about 4 to 5 representations of each of the characters that I wanted the network to learn.

I created a tool to generate the input characters during training. This tool allows the user to graphically input the character using a computer mouse, and tell the program what character is being represented. The following is an example of the input for the letter 'I' during training.

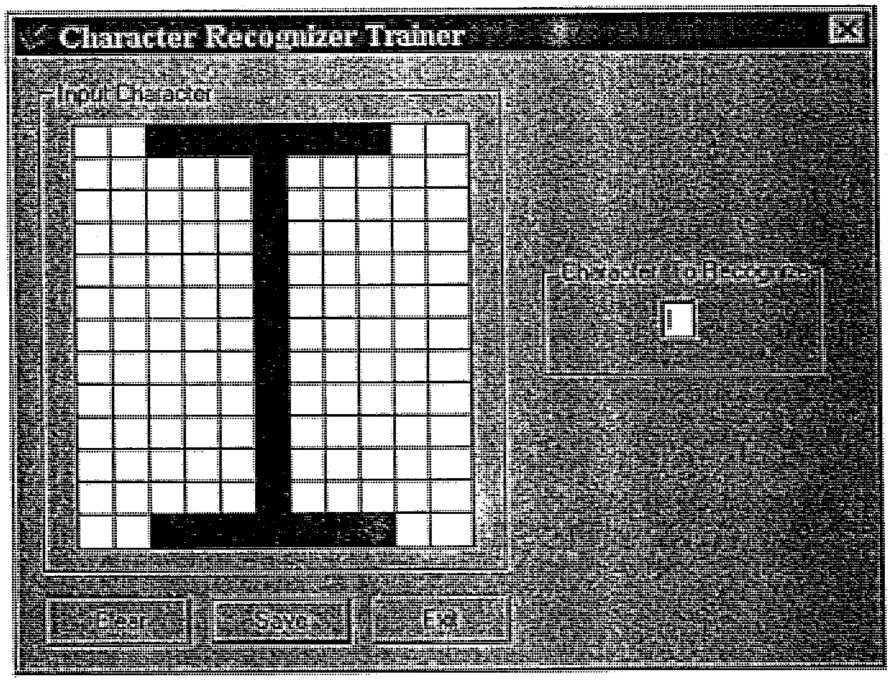


Figure 4: Training Graphical Interface

Each time the user clicks on the [Save] button, the character representation and its value is stored in a training file. Subsequent characters may be added by clicking on the [Clear] button to clear the input grid, and then enter the next character.

I used between 4 and 5 representations of each of the characters in my training set. Using 5 representations significantly improved the accuracy of the resulting character recognizer.

6. TESTING

I created a graphical user interface to test the Character Recognizer. Using this interface, the user can input a character using a mouse pad or any mouse device. The user can graphically see the letter they have input, and the letter that is recognized by the neural network. This graphical viewing of the results of the Character Recognizer made is very easy for me to test the application, and realize why unexpected characters were being recognized. Once I introduced 4 to 5 exemplars for each of the characters, the percentage of the characters that were being accurately recognized was over 90% even for some 'noisy' test cases. I tested the application with 5 person's handwriting, and the results were similar in all cases.

The following is an example of testing the result for the input character 'A'.

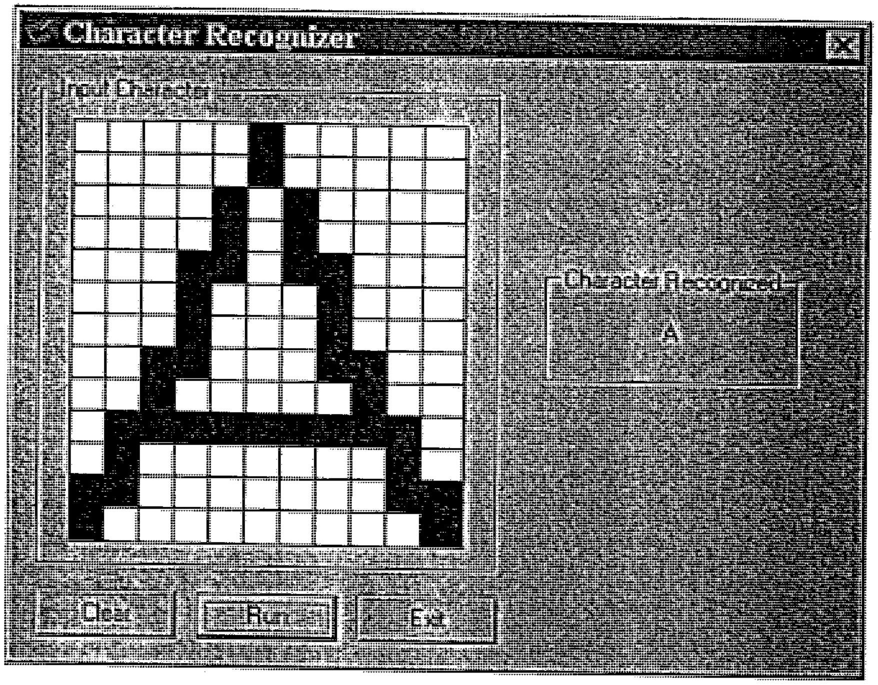


Figure 5: Testing Graphical User Interface

I ran a group of tests using the Character Recognizer inorder to see how well it performed among different amounts of noise in the input data. I carried out the tests by inputting each character about 20 to 30 times into the character recognizer, and tabulating the % of the time that the character recognized was what I actually meant. I inputted different amounts of noise by switching 10% and then 30% of the bits in the

1. INTRODUCTION

The search for a trading strategy which produces excess returns[2] (i.e., arbitrage as later defined herein) is the search for the modern equivalent of the "Philosopher's Stone"[3] of the Middle Ages. Claims about the ability to outperform a market portfolio -- without accepting risk in excess of normal market risk -- should be treated with extensive scrutiny and a sense of disbelief.

This paper presents:

1. A quick review of the development of the prevailing hypotheses regarding the efficiency of capital markets,

2. A brief description of several journal articles which suggest possible exceptions to market efficiency,

3. The formation of a hypothesis regarding one possible trading model, in one market,

4. A sampling of the literature on neural network trading systems, and

5. The training of a series of neural networks according to the proposed hypothesis.

2. THE EFFICIENT MARKET HYPOTHESIS

In 1900, the French mathematician Bachelier delivered his doctoral dissertation on the random movement of certain physical and financial events, such as the prices of financial assets. The paper was promptly forgotten, and his work was reinvented approximately 50 years later.

Von Neumann and Morgenstern (1944) published their work on game theory which was derived from a series of articles Von Neumann first published in the 1920's. The first part of their book, and its appendix, give an axiomatic treatment of individual utility. A key point of their analysis was that for measuring individual responsiveness, it is only necessary to achieve an ordinal (but not cardinal) ranking of final outcomes. In other words, we prefer A to B, but for many analytical purposes we do not need to know the units of measurement of this preference.

Markowitz (1959) developed the idea of the mean variance model of asset choice. This model assumes an individual preference for higher expected returns with lower mean variance, which flows from the assumptions about individual utility. In simple terms, we all seek to make the most money with the least risk.

Sharpe (1964) completes this picture of efficient markets with the Capital Asset Pricing Model which suggests that asset returns are correlated with asset risk. The model assumes that asset volatility (as expressed by Beta) is a good proxy for relative risk. Utility leads to portfolio diversification, which in turn leads to certain assumptions about how asset returns are related.

Some Notes on the Use of Neural Networks in the Financial Markets

Steven S. Strauss[1]
Yale School of Management
135 Prospect Street
New Haven, CT 06511
E-mail: Steven.Strauss@Yale.Edu

Abstract

The forecasting of financial markets has a long and distinguished history of failure. Certain individuals may have the ability to invest in such a way as to outperform the market (on a risk-adjusted basis). It is conjectured, however, that the fees they charge will almost certainly extract most of the excess returns (this hypothesis has been attributed to Paul Samuelson). The use of neural networks in the financial markets has recently received considerable attention. Most of the applications have focused on the larger Capital Markets (equities, options, futures) and seem not to have been integrated into any particular financial paradigm. The approach of this paper is to formulate an economic theory as to why an arbitrage should exist, and to utilize the neural network to test this tentative hypothesis. This paper differs from the existing literature on neural networks in finance by providing a more extensive financial hypothesis to justify the use of the proposed neural network. This paper also focuses on an application in a financial market (tax-exempt municipal bonds) which has not previously been explored using neural networks.

In the purchase of consumer goods, utility can be very hard to measure and the rationality assumptions above can be difficult to insure. In financial markets, however, utility and rationality are less tenuous assumptions. Most of us prefer $1,000,000 to $1,000, and would prefer a 50% chance of $200,000 to a 5% chance of $400,000.[4]

The very universality of these propositions contains the seeds of the efficient market hypothesis. Assuming all market participants have:

- approximately the same ordinal utility ranking,[5]
- the same access to information (no insider trading),
- sufficient capital and other resources --
- **Then markets should be efficient.**

An inefficient market would be one where a participant could borrow at one rate and invest at higher expected rates (without assuming additional risk) -- also known as an arbitrage situation. Given the assumptions above, if such an arbitrage opportunity existed, it would be bid out of existence. Assuming this reasoning is correct, all available information is accurately reflected in the price of financial assets. And if all information is accurately reflected in the price of an asset, it is not profitable to predict a future price.

This was the basis for the work done in the 1960's by Sharpe et al., and the various empirical studies which have been published. A similar line of reasoning stands behind Black Scholes' work on option pricing.

3. THE NON-EFFICIENT MARKET VIEW

A substantial financial money management industry exists, and why it exists is a bit of a mystery. According to the Reuters financial database, approximately 100 mutual funds (out of a universe of in excess of 6,000) were able to produce returns in excess of the S&P 500 at three, five and ten year intervals. Empirically this supports the efficient market hypothesis,[6] but also leads one to question some of our underlying assumptions regarding investor rationality.

Some academic studies have found evidence that markets are not as efficient as was originally postulated. In particular, Ross (1986) found evidence that industrial indices contain significant predictive power for stock price movements; Fama (1987) found that the Forward Yield Curve could explain approximately half of the future movement in short term interest rates; and Broughton and Chance (1993) found evidence that the Value Line Stock Rating service contains predictive power.[7]

3.1. Working Hypothesis

The working hypothesis of this paper is a weak form of market efficiency (i.e., that markets are efficient over time). Exceptions will occur in certain markets because of supply and demand factors, information lags, differential transaction costs, and technical limitations on the ability of market participants to utilize arbitrage opportunities. The markets most likely to be vulnerable to these problems should be relatively illiquid and hard to arbitrage (in the mechanical sense of executing the arbitrage transactions).

The United States tax-exempt bond market may meet these criteria. As the name implies, the yield on a tax-exempt bond is free from federal income tax. The yield on such a bond should be, and is, substantially lower than the yield on a comparable tax instrument.

Two key arbitrage techniques (short sales, and the traditional cash and carry) used by market participants to clear conventional markets are not available. Municipal securities cannot be borrowed and shorted.[8] Additionally, interest charges on funds borrowed to purchase tax-exempt securities are not deductible.[9] As a consequence the tax-exempt market may not have conditions for stable equilibrium's, and profitable future price predictions may be possible.

The yield on tax-exempt instruments should be influenced by:

- Interest rates on long term taxable securities such as the yield on 30 Year Treasury Bonds.
- Short term taxable interest rates such as 3 Month US Dollar LIBOR.
- Supply and demand for tax-exempt securities, where the proxy is new issue volume in the tax-exempt market.
- Current long term tax-exempt yields where the proxy will the Bond Buyer Revenue Bond Index (BBRBI), a commonly quoted index of long term tax-exempt securities.
- Income Tax Rates.
- The month of the year (if seasonal factors are important).

Regarding perceptions of future tax rates, I have no proxy measure for this variable. The absence of a measure for this variable is a significant flaw for the ability to make long term (beyond six months) predictions of market relationships.

3.2. Some Statistical Results

I have suggested some reasons for believing the tax-exempt bond market may be inefficient. Prior to training the neural network the factors listed above were tested (in a rudimentary manner) for signs of some statistical relationship.

Figure 1 Matrix of Correlation & Covariance Coefficients

Correlation

	Delta Libor	*Delta T30*	*Delta BB40*
Delta Libor	1		
Delta T30	0.246967913	1	
Delta BB40	0.26966818	0.619065806	1

Covariance

	Delta Libor	*Delta T30*	*Delta BB40*
Delta Libor	0.004200086		
Delta T30	0.000838329	0.002451242	
Delta BB40	0.000488533	0.000902043	0.000848487

The correlation's and covariance's were computed by calculating the daily change (e.g., the delta) in yield for a 1,500 day trading period for each of the indices. The correlation coefficient among the 30 Year US Treasury security yield and the long-term municipal bond yield (the BB40) suggest some validity to the theory that these interest rates are related. In addition, an inspection of the following chart suggests that these markets move in tandem.

input stream from 1 to 0 and then from 0 to 1. Surprisingly, even after adding 30% noise to the input character, the application still recognized most characters more than 80% of the time.

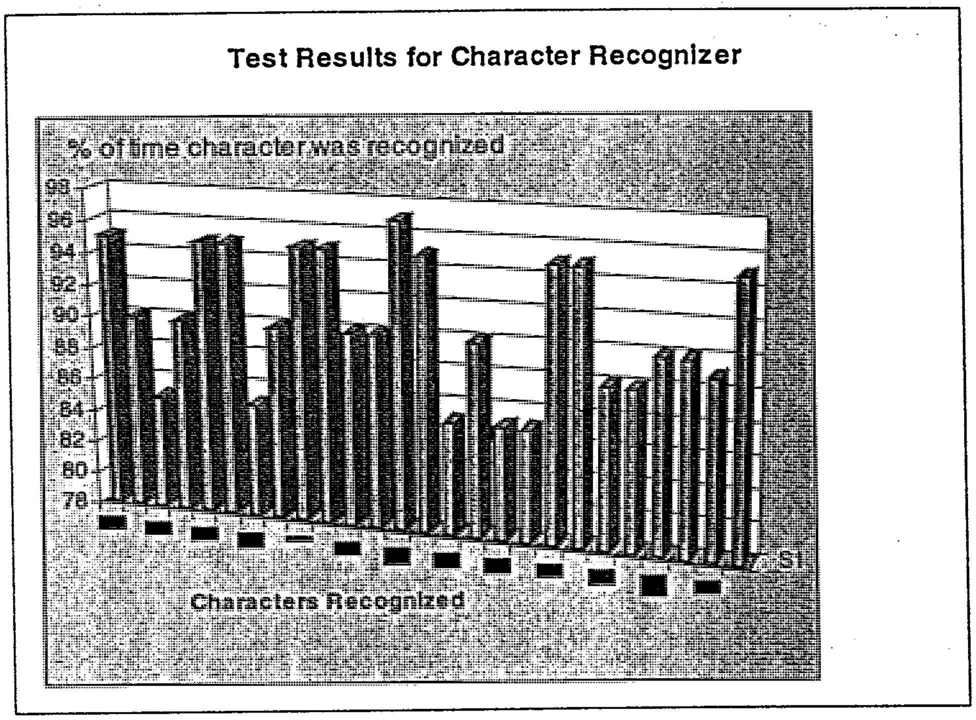


Figure 6: Test Results for Character Recognizer

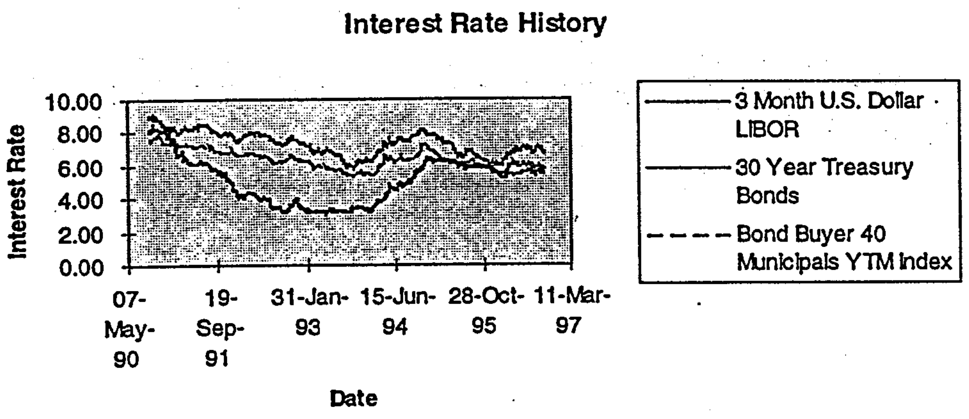
7. CONCLUSIONS

The application described in this paper shows an application in character recognition of neural networks using the backpropagation algorithm. The performance of the application is extremely accurate and reliable, recognizing evenly significantly noisy characters. The tools written for this application need not be used only for recognizing the letters of the English alphabet. Infact it can be used to recognize characters in any language as long as the inputs and outputs can be represented in a similar manner.

Characters are input into the neural network in this application via the mouse movements. This method of inputting characters can be improved by actually scanning written text to train the network.

Finally I would like to say that the results that I received after training the network were really interesting and exciting to me, since I only told the neural network what the correct solutions were during training, and not how it should learn what was correct. This learning mechanism in the backpropagation network really fascinates me, and I feel there are still several unexplored problems in the world that can be solved using neural networks.

Figure 2 Interest Rate Histories



We now have an economic theory as to why a relationship should exist between taxable and tax-exempt interest rates, together with some statistical observations which support the existence of this relationship. This brings us to a discussion of what the neural network will be trained to do.

Instead of training the network to predict the direction of interest rate movement, the network will be trained to predict the relationship between tax-exempt and taxable interest rates. The network will attempt to predict the future ratio of tax-exempt to taxable interest rates at 7, 30, 60 and 90 days.

Mechanically, we have no guarantee that market participants will be able to take advantage of the signals the network provides (for the reasons mentioned above), assuming the predictions are accurate. As a theoretical matter, the ability of the network to produce accurate relative value predictions (regardless of the real economic value) may have implications for future research.

4. NEURAL NETWORKS IN FINANCE

A number of authors have reported positive results using neural networks; among the most interesting (from my perspective) are:

- Malliaris and Salchenberger (in Trippi & Turban, 1996) who report good results using a neural network to predict options volatility,
- Trippi and DeSieno (in Trippi & Turban, 1996) who report good results using a neural network to track Index Futures.

Having reviewed a small portion of the literature, I believe that Refenes (1995) makes a highly significant point with his premise that neural networks are most effective when the creators have a correct economic theory. The network is used (under these circumstances) to solve for the details of the relationship. Refenes goes on to state:

"Approaches developed in statistical modeling should always be given serious consideration prior to training neural networks." Which is what was accomplished, in simple terms, in the prior section.

5. THE NEURAL NETWORK

The model is currently implemented using a Back Propagation Network ("BPN") in Thinks Pro, a commercially available Neural Network package.

BPN's are very commonly used for financial applications because of their generality. The BPN is considered to be highly generalized since it uses a gradient descent algorithm.

The gradient descent approach has the virtue of producing a closed form solution which minimizes the error between the input and output in the test data. The problem with this technique is that a BPN is "lazy" and the closed form solution may be of such a type that it is non-general and has no predictive value. For this reason, the input/output data is broken into a training set and a test set. The BPN is "taught" with the training set and tested with the second set.

A simple BPN algorithm, as described by Skapura (1996), is:

1. Select the first training vector pair from the set of training vector pairs.

2. Use the input vector as the output from the input layer of processing elements.

3. Compute the activation to each unit on the subsequent layer.

4. Apply the appropriate activation function.

5. Repeat steps 3 and 4 for each layer in the network.

6. Compute the error term across the output layer.

7. Compute the error for the hidden layer.

8. Update the connection-weight values to the hidden layer.

9. Update the connection-weight values to the output layer.

10. Repeat steps 2 through 9 for all vector pairs in the training set. Call this one training epoch.

11. Repeat steps 1 through 10 for as many epochs as it takes to reduce the sum- squared error to a minimal value.

This is the basic approach used by the neural network software. The activation function selected was the classic sigmoid function, no experiments were performed using other activation functions. A momentum term was used to speed the learning process.

Approximately 1,500 days of trading data were available. The data was divided into a training set and a testing set, each with approximately 750 days. The data set started on 8/30/96 and ended on 10/3/96, holidays and weekends were excluded.

For each day of training data the exemplar format was:

Figure 3 Sample Exemplar

Month	3 Month U.S. Dollar LIBOR (%)	30 Year Treasury Bonds (%)	Bond Buyer 40 Municipals YTM Index (%)	Visible Supply (in billions)	Ratio of Long Term Tax-Exempt to Taxable Interest Rates
8	6.13	8.98	7.64	3.84	0.85077951

"Month" is the month of the year, so for example "8" represents August. It has been speculated that the tax-exempt/taxable yield ratio exhibits seasonal fluctuations due to borrowing patterns and withdrawals of funds from the tax-exempt market to make quarterly income tax-payments.

The next three items are the yield to maturity of the appropriate index, on a given date. The visible supply captures the impact of changing supply conditions. It is the expected tax-exempt bond issuance for the next seven day period (in billions of US dollars), presumably unusually large supply should lead to lower bond prices (e.g. higher yields).

The last column is what the network will be trained to predict. This is the ratio of long term tax-exempt to long term taxable interest rates, as measured by the ratio of the index of the BBRBI to the U.S. Treasury 30 Year bond yield.

6. RESULTS OF NEURAL NETWORK ANALYSIS

In training the network the impact of the following dimensions were explored;

•Networks were trained to predict the tax-exempt/taxable ratio 7, 30, 60, and 90 days forward.

•The number of neurons in the hidden layer was varied from as few as 10 to as many as 200.

•Different network convergence algorithms were tested.

Approximately nine scenarios were analyzed in some depth[10], summary results for the scenarios with a seven day forward prediction are shown below:

Figure 4 Summary of Trained Neural Networks -- Seven Day Forward Predictions on Test Data

Filename	Nodes in Hidden Layer	Number of Training Iterations	Average error on Test Set	Maximum Error on Test Set	Exemplar with Maximum Error
NN07150	150	1,000	.032	.095	617
NN07175	175	1,000	.039	.229	739
NN07200	200	1,199	.040	.128	659

None of the scenarios involving making market predictions beyond seven days forward seemed promising. Upon reflections this also seems logical, as the value of technical information should be subject to rapid time decay.

The trained neural networks did not develop stable long term predictive power. In other words it was not possible to train the network, and allow it to continue making predictions, without retraining.

The networks seemed to develop an ability to make short term trend predictions, that may (with further refinement) have some economic value. In other words the network could predict (with some level of accuracy) the market ratio seven day's forward, but this power diminished.

Upon reflection this result is also not unexpected. Traders are constantly re-evaluating the value of information, and relationships in the market. In other words the human trader's biological network is retrained each day, so we should probably expect the same for the computer equivalent.

President Lehman Brothers, for comments and encouragement. The errors are, of course, my own.

[2] The standard unit of measure for financial calculations of this type is relative volatility to a market standard. As an example, the S&P 500 is considered to have a volatility level (Beta) of 1.0. The market portfolio should produce a return equal to the risk free rate plus the market premium. A portfolio that is twice as risky as the market should produce twice the return, but with much higher volatility. In other words in a market decline the high beta portfolio should decline faster than the market.

[3] The alchemist's vehicle for changing lead into gold.

[4] Actually even this is not as simple as it seems. A substantial body of work suggests people make "irrational" choices at extreme conditions. See Kahneman and Tversky (1979) for examples of choices people make when probabilities are low and payoffs are large. Another classic example is that the expected value of a lottery ticket is by definition negative, but the tickets are purchased on a large scale.

[5] This is a trickier assumption than would appear on the surface. The larger an agent's monetary base, the smaller the marginal utility of a loss or a gain, and the closer the participant to being classically risk neutral. In a financial market with one large and many small participants, the large participant may experience super-normal profits for a substantial period of time because of this difference in marginal utilities.

[6] An entire academic industry revolves around papers confirming the efficient market hypothesis. This small example is provided as an anecdote - not as a serious contribution to that branch of the literature.

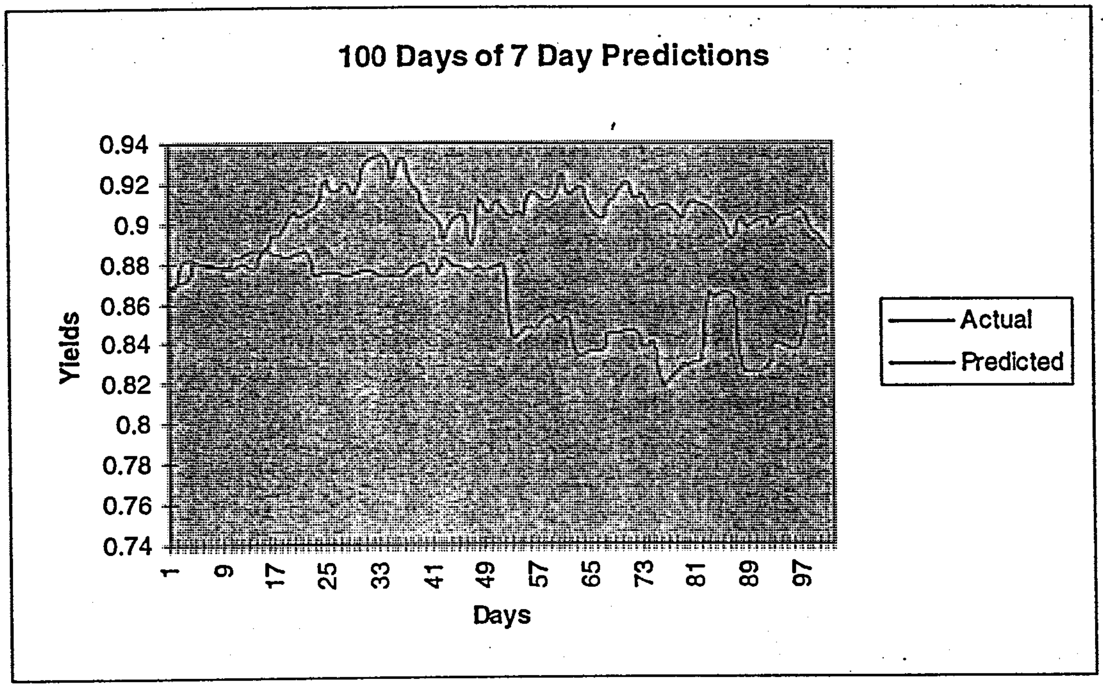
[7] The Value Line enigma has been studied, and confirmed, a number of times. Oddly enough the mutual funds managed by Value Line have not been outstanding performers.

[8] In general, when a security is borrowed and shorted the original owner receives a coupon, and the ultimate purchaser of the short sale receives a coupon payment. This cannot happen with a municipal bond because the result would be the creation of additional tax-exempt interest.

[9] If interest could be deducted, it would be the ultimate arbitrage. A purchaser could purchase a tax-exempt bond, deduct the interest, collect the tax-exempt coupon, and be guaranteed a profit.

[10] As a side point each scenario took between 3 - 6 hours to be trained on a Pentium - 100.

Figure 5 Predicted vs. Actual Results for 100 Trading Days



The results shown above are for one of the networks, trained on 744 days of data, and tested on the sequentially following 750 days of data. Note that the network has some gross predictive ability for approximately the first two weeks, but this ability declines rapidly. This result was apparent in other scenarios, as well.

7. CONCLUSIONS

I am not particularly discouraged by these results. Indeed, I would have been surprised if I had easily found an exploitable arbitrage, given the number of full-time professionals looking for these types of relationships. The power of Neural Networks to fine-tune known relationships is an under-exploited resource with substantial business applications. Applications of neural networks to learn about customer buying habits and to suggest ways of optimizing marketing programs seem to be worth further investigation. The areas of predicting financial market results appear to be well-covered, and I continue to have theoretical reasons for doubting long-term claims to success in the more liquid capital markets. At the same time I believe these types of models may have substantial possibilities in less liquid markets, and as automated clearing house systems in place of floor brokers/traders.

References

1. Brealey, Richard A. (1981). *Principles of Corporate Finance.* New York: McGraw-Hill
2. Broughton, John B. & Chance, Don M. (1993). The Value Line Enigma Extended: an examination of the performance of option recommendations. *The Journal of Business,* v66, n4, p541(29)

3. Chen, Nai-Fu, Roll, Richard & Ross, Stephen A. (1986). Economic Forces and the Stock Market. *The Journal of Business*, v59, n3, p383(21)
4. Fama, Eugene F. & Bliss, Robert R. (1987). The Information in Long-maturity Forward Rates. *American Economic Review*, v77, n4, p680(13)
5. Haykin, Simon S., (1994). *Neural Networks: a Comprehensive Foundation.* New York: Macmillan
6. Kahneman, Daniel & Tversky, Amos (1979). Prospect Theory: An Analysis of Decisions Under Risk. *Econometrica*, Vol. 47, No. 2 pp. 263-292.
7. Makowitz, Harry M. (1959). *Portfolio Selection.* London: Basil Blackwell
8. Peters, Edgar E. (1994). *Fractal Market Analysis: Applying Chaos Theory to Investment and Economics.* New York: J. Wiley & Sons
9. Refenes, Apostolos-Paul (editor) (1994). *Neural Networks in the Capital Markets.* New York: John Wiley & Sons
10. Sharpe, William F. (1964). Capital Asset Prices - A Theory of Market Equilibrium Under Conditions of Risk. *Journal of Finance*, September 1964, pp. 425-442.
11. Skapura, David M. (1996). *Building Neural Networks.* New York: ACM Press
12. Trippi, Efraim Turban, (editors) (1993). *Neural Networks in Finance and Investment : Using Artificial Intelligence to Improve Real World Performance.* Chicago: Probus.
13. Von Neumann, John & Morgenstern, Oskar (1945). *Theory of Games and Economic Behavior.* Princeton: Princeton University Press

Appendix A: Profit Opportunities

The output of this network predicts neither future interest rates nor the expected price of a particular security, but rather the ratio of the yield between two markets. The purpose of this appendix is to present a simple explanation of how this information could profitably be put to use. Please note this explanation is not intended to provide detailed trading advice, but simply an outline of the approach for the non-financial reader.

As a starting point, assume in time period 0, tax-exempt bonds are trading at a yield to maturity of 8.50% (for some maturity). U.S. Treasuries are trading at a yield of 10.00% to the same maturity. The yield ratio is 85%. Also, assume that the neural network model is predicting that this ratio will reach 50% in time period 1. The ratio can change by an increase in tax-exempt yields or a decrease in taxable yields, or some combination of both.

If our hypothetical investor purchases $850,000 in tax-exempt bonds, and shorts (sells securities that are borrowed) $1,000,000 in U.S. Treasury bonds, she will profit so long as the predicted ratio is achieved, no matter which path is taken. For example, if the 50% ratio is achieved because tax-exempt interest rates remain unchanged, and taxable rates increase to 17%, the result will still be a profit.

The profit (in this case) will result from closing out the short sale. The increase in taxable interest rates means that the Treasury bonds that were sold in time period 0, can now be repurchased at a substantially reduced price. This sale and repurchase (in this scenario) produces the profit.

[1] This paper was prepared as part of course CS577a (Neural Networks in Computing) for Prof. W. Miranker, at Yale University. I would like to thank Prof. Miranker and Robert Taylor, Senior Vice

Principal Component Analysis for Place Recognition

Jonathan Wang, Zachary Dodds, Willard Miranker[1]
Computer Science Department, Yale University
New Haven, CT 06512

Abstract
We present a hybrid neural network model to solve a place recognition problem. The front end is a self-organizing net equivalent to a principal component analyzer; the back end is a feed-forward net with backpropagation, i.e. supervised learning. A confidence level greater than 0.9 was reported as the net correctly recognized a repertoire of pictures it had not seen before.

1. INTRODUCTION

At the Yale Vision and Robotics Lab there is a Nomad robot that roams around the building taking pictures. It is desirable that the Nomad can recognize where it is by comparing a new scene with previously taken pictures. Formally, suppose we have images of M distinctive scenes. (We will use the terms "picture" and "scene" as synonyms for "image" throughout.) We seek an algorithm that will take a new image as input and determine which one of the M scenes the new image most resembles.

Here, we propose a neural network solution which combines stages of unsupervised learning and supervised learning. The network is composed of two independent subnetworks. The first subnet, which we call the "Principal Component Analyzer" (PCA) is self-organizing. It receives an image (the "input image,"possibly preprocessed) and outputs a set of real numbers. The latter express the most "important components" in the image. These "components" are the longest axes of the ellipse which bounds all of the images considered as data points in an appropriate, high-dimensional space. The second subnet, which undergoes supervised learning, is a feed-forward backpropagation network. It takes as input the output coefficients from the first subnet. It outputs, in its M output nodes, a confidence measure ([0,1]) indicating the extent to which the input image corresponds to each of the M distinctive scenes.

In Section 2 we discuss the PCA net, its theory and the algorithm behind it. Section 3 deals with the backpropagation net. We conclude in Section 4 with some observations and comments on future directions that this work suggests.

2. PCA NET

The PCA net is a single layered network of n inputs $X = [x_1, x_2, \ldots, x_n]^T$ and m outputs $Y = [y_1, y_1, \ldots, y_m]^T$. (See Figure 1.)

Each input x_i corresponds to an image pixel. Here,

$$n = (\#\ of\ image\ rows)(\#\ of\ image\ columns).$$

[1]Research Staff Member Emeritus, IBM T.J. Watson Research Center, Yorktown Heights, NY

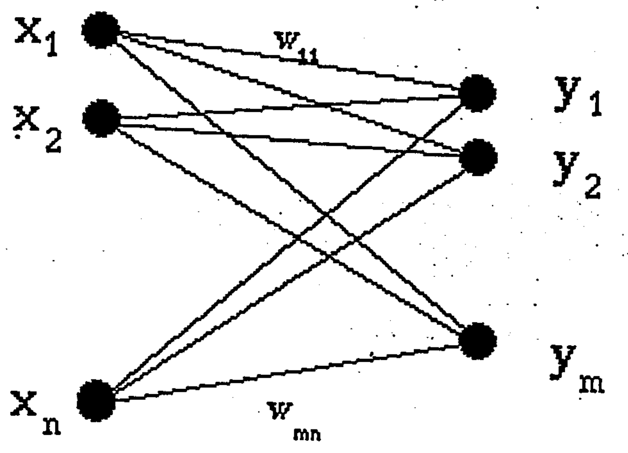


Figure 1: **Diagram of the PCA Net.** The nodes on the left, x_i, receive the gray-level values of the input image pixels. They are connected to output nodes, y_i by weights w_{ij}. The output nodes yield coefficients of the principal components for the given input image.

In our experiment (64x48)-pixel images are used, so that $n = 3072$. We can view the input "vector" by "stacking" columns of an image to form a 3072-component column vector. Since our camera and framegrabber return integral gray-level values between 0 and 255, those are the minimum and maximum values for each x_i.

The strength of the connection between the input x_i and the output y_j is given by a weight w_{ji}. In particular, we have

$$y_j = \sum_i w_{ji} x_i.$$

We use the "Generalized Hebbian Algorithm" (GHA) as the training algorithm. The dynamics specifies the weight of a connection after update (written w_{ij}^{+}): (Sanger, 1989)

$$w_{ij}^{+} = w_{ij} + \gamma(y_i x_j - y_i \sum_{k \leq i} w_{kj} y_k).$$

Setting $W = (w_{ij})$, we express this in matrix form as:

$$\Delta W = \gamma(YX^T - LT[YY^T]W),$$

where $LT[\cdot]$ sets all elements above the diagonal of its matrix argument to zero, thereby making it lower triangular. The learning parameter, γ, specifies the rate of learning and influences how quickly the weights converge and if they converge at all.

We first state, without proof, a convergence theorem. (See Haykin, 1994, Appendix B for a proof.) In the following sections, we explain the related concepts.

Theorem Let the components of W be assigned random values at time zero. Then, with probability 1, W will converge to the matrix whose rows are the first m eigenvectors of the input correlation matrix $Q = E[XX^T]$, ordered by decreasing eigenvalue. (Note that Q is a symmetric matrix.)

2.1 Principal Component Analysis

The motivation behind the GHA algorithm is to compress data and to preserve as much of the information in the input as possible. For example, in our implementation we reduce a 3072-pixel (i.e., 3072-dimension) image into a 4-dimensional vector. Principal Component Analysis allows us to find the four dimensional vector which captures the most variance (which we may view as a measure of information) in the original data.

Let X be a random variable each component of which has zero mean. (A change of variables ensures this in the general case.) Consider the collection of all possible images as the sample space of the random variable. The autocorrelation matrix Q of the input signal distribution is defined by

$$Q = E[XX^T],$$

where E is the expectation operator. Let u_i and λ_i $(i = 1, 2, \ldots, n)$ be the orthonormal eigenvectors and the corresponding eigenvalues of Q, respectively, the latter taken in decreasing order. Define the corresponding matrices

$$U = [u_1, u_2, \ldots, u_n]$$

and

$$\Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}.$$

Eigenvectors u_i of Q are called "principal components" in signal processing (hence the name of the net). If the eigenvalues are distinct,

$$Q = U\Lambda U^T.$$

By using u_i as basis vectors, a given image, X, can be expressed as linear combinations of those basis vectors, as follows.

$$X = \sum_{i=1}^{n} u_i y_i = UY.$$

So that, for the coefficient vector, we have

$$Y = U^T X.$$

Since U is unitary, $U^{-1} = U^T$. The coefficient y_i is thus the magnitude of component u_i contained in X.

Put another way, typically there is much redundant information in a raw image. Except for rare cases (e.g., the noise of a "snowy" television channel), the pixels in the image vector, $X = [x_1, x_2, \ldots, x_n]^T$ are "correlated". By this we mean informally that the gray-level values at some points in a picture are predictable from these values at other points in the same picture - all of the pixels in a full moon, for example,

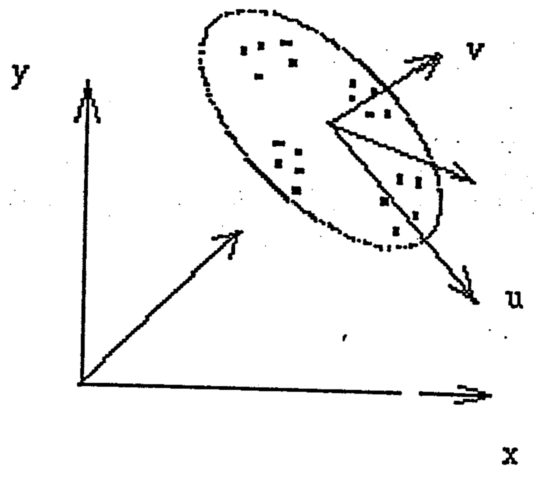


Figure 2: **Principal Components.** The principal components are the semiaxes, u and v, of an ellipse which contains the data points.

will be close to white. To obtain a more compact representation of the information in the image, we seek the above transformation U^T. The latter ensures that $Y = [y_1, y_2, \ldots, y_n]^T$ has *un*correlated components. By a theorem of Karhunen and Loeve (Rosenfeld & Kak, 1981) the transform matrix U^T consists of the eigenvectors of the autocorrelation matrix Q. The weights of the PCA net converge to these eigenvectors, so that the net's output is uncorrelated. Since the redundancies in the inputs are removed, the output variance will be maximized. The outputs represent the largest possible amount of information which a fixed, small number of dimensions (four, in our case) can convey. It might be easier to understand the geometry of principal components through Figure 2.

In the figure xy–space is a n-dimensional vector space. Each point in xy space represents an image (that is, represents the n-dimensional vector which corresponds to the gray-level values of the pixels in an image). The basis for xy–space is orthogonal, but might not represent the information contained in the images as efficiently as possible. As we noted, the projection of each image-as-a-vector onto those basis vectors will be correlated (with redundant information). We seek a new set of basis vectors that are mutually orthogonal and which better represent the variance of the data vectors. We proceed as follows: We find a best-fit n-dimensional ellipse (or ellipsoid) around the vectors. The largest axis of the ellipsoid is the first principal component, the second largest is the second principal component, and so on until the nth largest axis is the nth principal component. We can see that the new basis vectors, as semiaxes of an ellipse, are indeed orthogonal to each other. As it turns out, the new basis vectors are the principal components of the images, which are the same thing as the eigenvectors of the correlation matrix. (Rosenfeld & Kak, 1981)

2.2 Advantages and Disadvantages

From these considerations we see the significance of the GHA algorithm. The dynamics defining ΔW define the principal component vectors in question. Moreover, the vectors need not be computed, as the net itself instantiates the dynamics and the

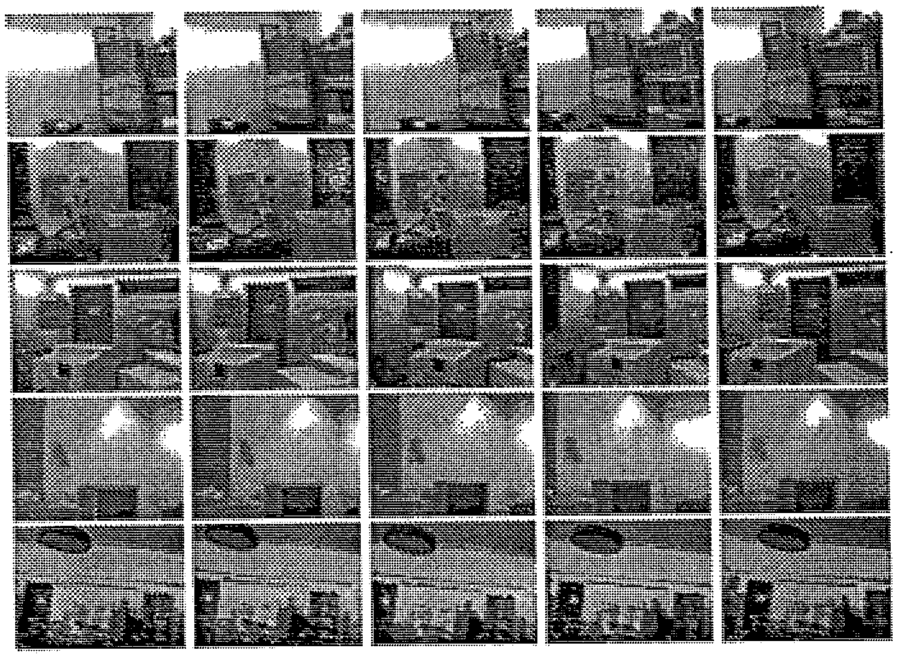


Figure 3: Training Images. We chose five different views of each of five parts of the Vision Lab at Yale. The four most important principal components are extracted and the backpropagation net is trained on the coefficients of those components.

appropriate subsequent pixel processing. For $n = 3000$ input nodes, $Q = E[XX^T]$ has 9 million components. If the number of outputs is much smaller than the number of inputs, as in our case, GHA finds the most important eigenvectors - that is, the eigenvectors with largest eigenvalues without having to decompose the huge matrix Q. In addition, GHA is a neural net algorithm with the potential for high-speed, special-purpose hardware.

A disadvantage is that GHA provides only an approximation to the eigenvectors. Furthermore, as in all such numerical methods, errors in the first few eigenvectors will magnify the errors in the subsequent eigenvectors, so that the algorithm has poor numerical accuracy for all but the first few eigenvectors. For our images, the number of samples, N, is very small compared with n, the dimensionality of the space where the samples are drawn. Therefore, we require only the first few eigenvectors (the principal components). Note, too, that the algorithm only involves local operations. Hence, it is possible to implement GHA on a parallel machine, though the communication of data will require an overhead of time.

2.3 Experiment

The PCA net is trained with $N = 25$ images, five each of five distinctive scenes from Yale's Vision Lab. (See Figure 3). Each image is input to the net 60 times.

GHA is used as the learning rule to adjust the weights of the network. Although normalization of input does not affect the output of the net, it does influence the choice of learning rate γ. (Experiments with power spectra of images as input, for instance, will require a different learning rate.) We take an experimental approach to specifying a good value of γ. Too large a γ will drive the net to saturation, where the values of the connections are outside of the representable range of the computer, and too small a γ will make the net take too long to converge. For our net with 256-

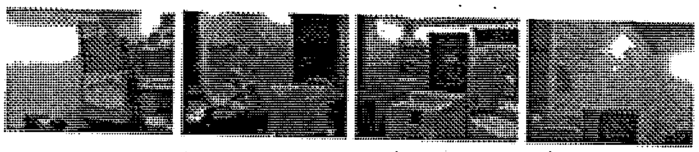


Figure 4: **Test Images.** These are four different views, one of the five views above is omitted. We provided these as inputs to the system after it had been trained on the images in Figure 3.

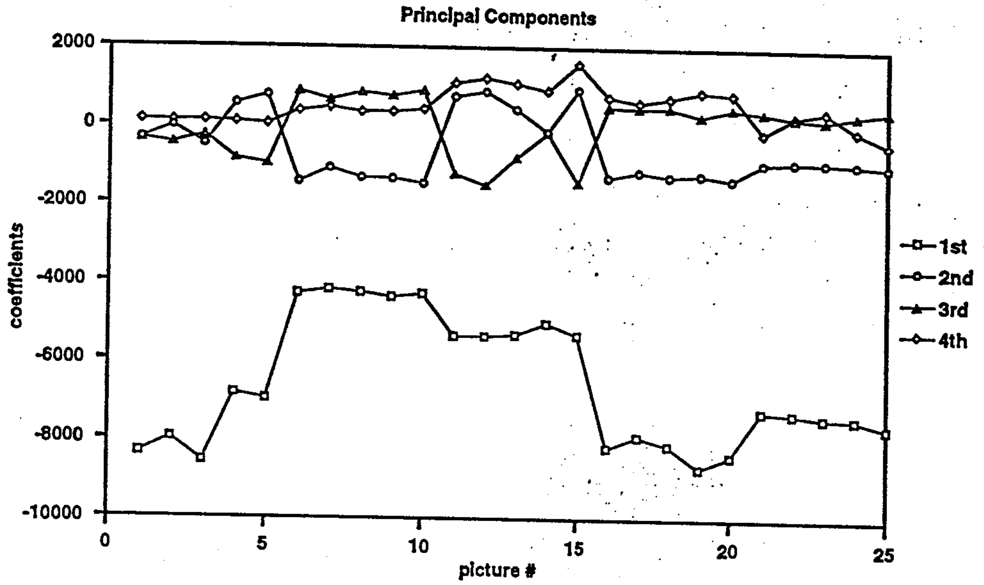


Figure 5: **Coefficients of the Principal Components.** The values of the coefficients of the principal components obtained from the PCA net on the twenty training images. They are grouped in fives: 1-5, 6-10, 11-15, 16-20, and 21-25. Each such group contains five slightly different images of the same scene.

level gray-scale images as input, γ was set to 10^{-8}. Several observations were made while tuning this learning rate. First, γ should decrease with time. Second, different output nodes should use different γ values, since each coefficient (node) converges with slightly different rates. We employed a purely empirical approach to this tuning of the γ values.

Once the net converges, we have found - to some accuracy - the principal component vectors for our data set. Those vectors are stored as the weights of our net. Next, we freeze the weights and input the training set again. For each such input we obtain the coefficients of those principal components, one coefficient at each output node. The coefficients from our images are plotted in Figure 5. These coefficients are the values of the projection of the N vectors (images) onto the first M (in our case M = 4) principal components (the basis in uv-space).

In Figure 5 the first five picture numbers (labeled on the horizontal axis) correspond to the same scene, as do the following five, the third set of five, and the last five. The flatness of each of the five curves shows that images of the same scene have similar coefficients. We represent, then, a location by the coefficients which are characteristic of images taken of that location. Suppose that an arbitrary set of coefficients is specified. We need to decide how much that set of coefficients resembles the sets which have been stored. If the coefficients of a new scene are sufficiently close to one of the stored sets, we will conclude that the new scene is the same as the one in our database. In this way we are able to decide whether or not we are at a previously-seen location, as well as which such location. We decided that, in principle, any such decision requires a teacher. We chose to teach a feedforward net the classifications we wanted; in a sense, the training of that feedforward net will provide the idea of "sameness" that we intuitively feel.

3. BACKPROP NET

We did not apply backpropagation directly to train the feedforward net to identify scenes – the most salient reason being that backprop is slow to converge. With a net of 3072 input nodes there would be

$$3072(\textit{number of hidden nodes}) + (\textit{hidden nodes})(\textit{number of output nodes})$$

free parameters to tune, and we have no clear heuristic to guide the tuning.

In addition, human scene recognition involves considerable preprocessing – edge detection, noise reduction, feature extraction, etc. It is therefore not surprising that a stand-alone backprop net converges slowly, since the complexity of that preprocessing must be expressed in the net's weights. Hence, it is desirable that we separate some preprocessing into a separate system and reduce the number of the free parameters in the backprop net. Biological results have shown that the equivalent of a principal component analyzer exists in receptive fields. (Rubner & Schulten, 1990)

We take it that a place recognition system can not rely solely on the unsupervised learning of the PCA net. Consider, for example, a Hopfield Net acting as a memory, and which does not require supervision or other outside judgment of its performance. In Figure 6, we show that a trained Hopfield net when stimulated by a slight variant of one of its training samples, does not give a correct recall. Rather, it outputs an image that doesn't resemble any of the training samples, a so-called spurious state. It may be that the problem of spurious states does not stem from the architecture of a Hopfield net, but rather from a fundamental lack of information. The ability to recall a "noiseless" version of a "noisy" stimulus is attributed to a Hopfield net. Just what is a "noisy" stimulus? How can it be distinguished from a new "noiseless" stimulus or - even worse - from a new, but different, "noisy" stimulus? The "spurious" state is actually a correct recall, in so far as the Hopfield net is concerned. These questions motivated the conclusion that wholly unsupervised learning is not appropriate for place recognition. We want machines to perform a classification similar to the ones we, as sentient agents, agree on. Some sentient intervention is thus required. We must somehow impart the assumptions which we internalize to any system which we want to act as we do. Simply put, we found it effective to introduce supervised learning in the place recognition problem.

3.1 Experiment

The "art" of creating a backpropagation net lies in choosing how many free parameters we permit the system to learn. If we have too few free parameters (by having too

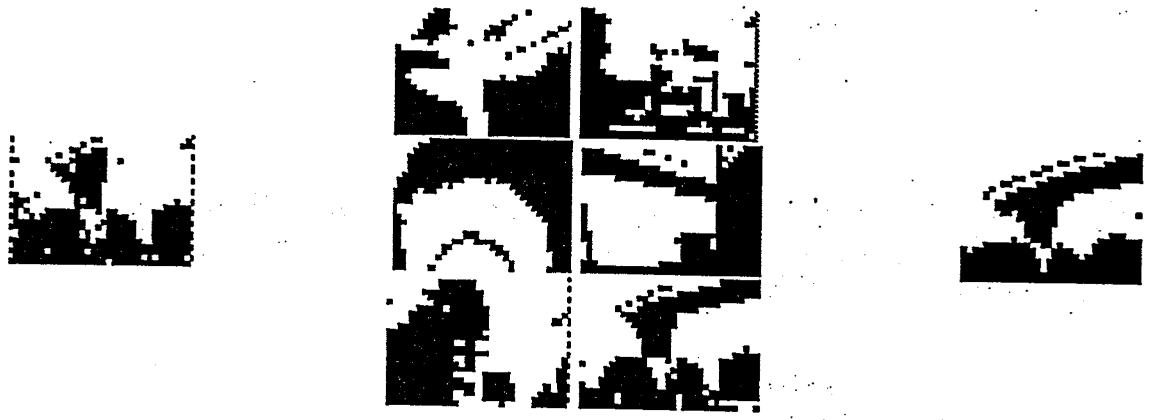


Figure 6: **Hopfield Net.** The middle columns are the training set. All are low-resolution photographs. The left column is a noisy version of the lower right training sample. The right column is the erroneous recall, i.e., a spurious state.

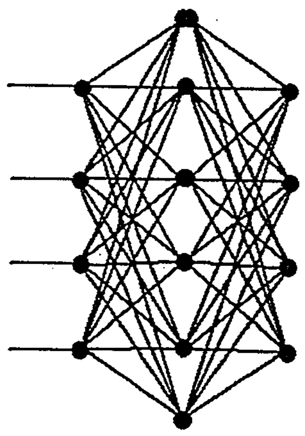


Figure 7: **Backprop Net Schematic.** The left column of neurons receive the inputs from the PCA net. The middle column comprise the hidden nodes. The right column is the five output nodes, one per location.

few hidden nodes) then we will not be able to capture the complexity of the classes. The XOR problem, as a fundamental example, is not solvable by a backprop net without hidden nodes. If we have too many free parameters, the learning will require longer to converge. Worse yet, the net will "overfit" the sample data it receives in its epochs of input. While it will learn the exact input-output mapping we teach it, that mapping will not generalize well to other, unseen inputs. Empirically, we found that a backprop net with 6 hidden nodes sandwiched between 4 inputs (one for each of the four principal component coefficients) and 5 outputs (one for each of the five places) worked well for our problem of recognizing lab scenes and, in addition, converged quickly.

The net was initialized with small, random values for weights between layers. The transfer function is the sigmoidal

$$y_j = \frac{1}{1+e^{-v_j}},$$

$$v_j = \sum w_{ji} u_i,$$

where y_j is the output of neuron j and u_i is both the output of neuron i and the input to neuron j. w_{ji} is the weight connecting the ith input to neuron j itself. The activation level of neuron j is v_j, which is the weighted sum of the inputs to that neuron.

The inputs are presented to the network in epochs, in each of which all N inputs (N = 25 in our case) are processed. The five output nodes in each case yield values in the interval (0,1). The "correct" value, i.e., the one we want the network to yield is a five-tuple with a 0 in each of the components which corresponds to an incorrect location and a 1 in the component which corresponds to the correct location. Of course "correct" and "incorrect" are determined by human observers. Note that the network can not output the precise expected value, since the range of the sigmoidal transfer function does not contain 0 or 1. Thus, the weights will diverge toward infinity, as they drive the sigmoidal function toward its limiting values. We do not use a step function, since we desire the continuum of output values. The range (0,1) provides a value for interpretation as "confidence" in the recognition. Our network consisted of four inputs, taken from the four outputs of the PCA net. It also contained six nodes in its middle layer and five outputs, one for each of the five scenes the system was meant to recognize. We stopped the training after 20,000 epochs, which required approximately 30 seconds on a Sun Sparc IPX.

When the net outputs a vector, that output is compared to the desired result, and the weights are adjusted in the direction of the local gradient throughout the network – the standard backpropagation algorithm. The learning rate and momentum parameter can be adjusted to speed the convergence of the net and avoid sending the coefficients off to infinity. (We found that a learning rate, $\eta = 0.1$ and a momentum parameter, $\alpha = 0.3$ performed well through experimentation.) The weights w_{ij} are frozen after training to store the classifications the net has just learned. Finally, the unseen test images are input to the net, and the outputs are compared to the desired outputs. The following table yields the results of the five input test images.

No.	*Output 1*	*Output 2*	*Output 3*	*Output 4*	*Output 5*
1	.939	$4.12 * 10^{-4}$	$1.10 * 10^{-2}$	$2.82 * 10^{-4}$	$7.18 * 10^{-2}$
2	$6.96 * 10^{-5}$	.946	$1.91 * 10^{-2}$	$6.67 * 10^{-2}$	$1.38 * 10^{-2}$
3	$3.19 * 10^{-2}$	$3.65 * 10^{-3}$	.979	$2.09 * 10^{-2}$	$1.20 * 10^{-5}$
4	$1.99 * 10^{-4}$	$5.81 * 10^{-2}$	$5.31 * 10^{-2}$	.921	$5.34 * 10^{-2}$

3. OBSERVATIONS AND CONCLUSIONS

It is well-known that the human vision system has a layered structure. There have been attempts (Leen et al, 1992, Dumont & Yang, 1991, Metcalfe & Cottrell, 1991) to decompose the difficult problem of recognition into subproblems and to use different neural networks to solve each subproblem. Our PCA net and backprop net work in a similar fashion, i.e., the PCA net is a preprocessor (Principal Component Analyzer) for the backprop net, which is the classifier. We could also extend this a few more stages. For example, we could have preprocessors to extract features, such as edges. Then, instead of using raw images as input, the PCA net could work on an

The training of the network is a repetitive process of giving the net an input, and then letting it run to settle down on its weights, and then giving it the next input. If the net converges after processing all the training samples, its weights should be stable, and the information is considered to be stored in it.

The recall process is similar with the training process, which has strong analogy to the biological systems, except that in practice, the weights are often frozen before recalling process for simplicity. After the testing input, the network should settle down on some output. It could either be a previous seen pattern of which the input is a noisy version, in which case the input is considered to be correctly recalled, or it could produce a wrong answer by retrieving a different previous seen pattern. A third possibility is the returning of a spurious pattern. For correct recalls, feasible memory is a necessary condition.

2 Program Design Issue

2.1 Hebb Constants

There are four constants in the Hebbian formula used in the system: $H_{ij}^{uv} = {a_0}^{uv}xy + {a_1}^{uv}x + {a_2}^{uv}y + {a_3}^{uv}$. The association rules for excitatory and inhibitory connections are different.

For excitatory connections, the weights should be increased when there is positive correlation between the two activities, and should decrease when the two synaptic signals are anti-associated or non-associated. Therefore we chose the criteria for a_0, a_1, a_2 and a_3 to be:

$$a_0 + a_1 + a_2 + a_3 > 0$$

$$a_1 + a_3 < 0$$

$$a_2 + a_3 < 0$$

$$a_3 < 0$$

The last rule is not required by the Hebbian dynamics. We required it since it worked better when we maintain $a_3 < 0$ during our experiments. This is probably due to punishment of non-associative signals.

For the inhibitory weights, things are bit less clear. The base line is that an outgoing inhibitory weight from a neuron whose output is 1 should increase, since in the next round a stronger signal should be sent to its neighbors to inhibit them. But for the nodes whose output are zero, we could either decrease the outgoing weights, or leave them alone. Since we generally prefer to balance our weights adjustments to consist of both increasing and decreasing movements, we prefer to decrease a outgoing weight for a no-active neuron. This consideration is summarized by the following formula:

$$b_0 + b_1 + b_2 + b_3 \leq 0$$

$$b_1 + b_3 < 0$$

$$b_2 + b_3 > 0$$

$$b_3 < 0$$

Note these rules are not symmetric for x and y. This is because $H_{ij}^{uv} = a_0{}^{uv}xy + a_1{}^{uv}x + a_2{}^{uv}y + a_3{}^{uv}$ is the change of weight for the synapse from neuron $n^v{}_j$ to $n^u{}_i$.

2.2 Initial Values

The rest of the initial parameters are set to random values. v', the output are randomly set either 0 or 1, and u', the input takes a small random variable from zero and one. The thresholds are chosen to be positive but less than one too. Originally we set w_u and w_l to be 1.9999 and 0.0001.

2.3 Stopping Criteria

We do not wait until convergence of the weights to stop. In fact, at each training input, the program runs a certain number of epochs, and goes for the next input. After training on a certain number of samples, we perform the recall. When we find a reasonable recall, we examine the weight changes of the network to see whether a convergence has arrived. Most of the time for the two nodes per layer case, when we get the correct output, the net is actually converged. But for larger nets (with 3 or more neurons per layer), the convergence is very rare, as we will discuss later.

3 Experiment Results

3.1 Training Protocols

We started from the simplest case of the network: two nodes in each layer. We are able to successfully train our net when there are only two nodes on each layer. We achieved this by adjusting the parameters for the Hebb rules progressively.

First, we noticed that there is a trivial solution for the net when the net will produce the exact input as the output. This is when all of the weights are zero. Since we are observing the output from the 0th layer, when the weights are zero, any internal presynaptic and postsynaptic signal will not influence the output at all. Only the input signals will affect the output of that 0th layer. This is probably because our thresholds are less than one, all the inputs are either 0 or 1, and the step function (our activation function) will filter out all information which is not big enough.

However, the trivial solution is not what we want. We strengthened the Hebbian rewards for positive correlations while at the same time gives more positive examples (here I mean using training samples which have more bits "on" since those "on" bits are the positive information to be processed by the net). And we carefully adjusted other parameters. The memory starts to give us correct results.

In one experiment, we first train the four nodes net only by inputting [1, 1]. After it gives the "correct" recalls for every possible input (by responding output [1, 1]) and the weights converge fairly well to some non-zero points, we give it another input say [0, 0]. After a few training epochs, the net can correctly recall [0, 0] and [1, 1], and also classify [0, 1] and [1, 0] as a noisy cue of [0, 0] and [1, 1] without giving any spurious results.

In the third step, we train the net with all four possible inputs. After adjusting the training parameters, we can successfully obtain all four correct recalls. We examined weight changes, and it seems that the network settles down very well. Figure 1 and figure 2 show the changes of two of the weights.

Although we were able to determine successful parameter settings for all of these different cases, we were not able to find a single set of parameters which gives good results in all cases. We think this is because it is much easier to find a special net for a particular training task. To find a general net is much harder, especially when the Hebb rules that we used are linear

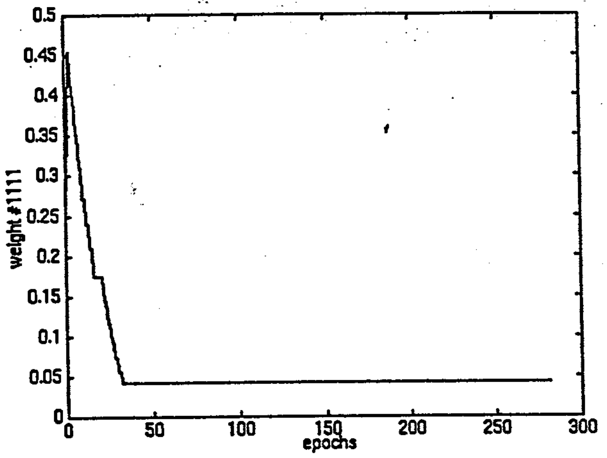


Figure 1: The change of weight #1111

and sensitive to local information (no normalization or adjustment are used).

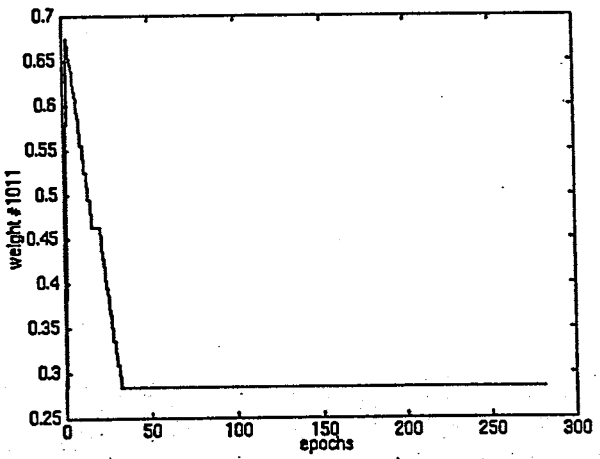


Figure 2: The change of weight #1011

3.2 Net With More Than Two Nodes Per Layer

For the nets which have 3 or 4 nodes per layer, the results were not as clear cut as what we have talked about. Using an exhaustive search method, we did find some parameters which can train to net to give correct recalls. However, when we examined the weight changes. It seems that the weights are still oscillating when we stopped training. We suspect that the correct output is only a coincidence of weights. However, it might be one of the feasible memories of the net that we have found, but we simply don't know whether it will conserve feasibility property or not.

This difficulty is anticipated, since the weight searching space grows quadratically as the number of nodes increases. Therefore finding the route from a random point to a feasible memory is more difficult and the process should encounter more attractors, even though there might be more than one feasible memories.

3.3 Learning Rate

We tested different learning rate's influence on the training of the net. Smaller learning rate helps in getting give better convergence since it will reduce the magnitude of the oscillations, and help the weights to settle on a set of values. But with learning rate smaller, we need to give it more epochs per training sample, since the weights will take more steps to complete a transition.

4 Self-organization with Maximum Eigenfilter Formula and PCA

In our previous training of the net, a very frequently encountered problem is the inappropriate parameters will make the weights to go against lower and upper limit. Therefore we have to truncate the weights to at the upper or lower bound. Doing so, we lose some information. One remedy is to introduce some global information and change the δw's calculation. A good candidate is the maximum eigenfilter model. We modified our weight updating formula to be

$$w(n+1) = w(n) + \eta y(x - yw)$$

for our δw.

The $\eta y x$ part represents the usual Hebb rule, and $\eta y y w$ is a dissipative correction term for stabilicy. The same formula has been used in one layer self-organized principal components analysis. One hope of ours is in our architecture this formula will lead to similar results.

This modification indeed helped the convergence of weights. We plotted the net's weight changes during any training period, and find that the wildly oscillating disappears. (Please see attached plot.) However, we haven't shown the relationship of the eigenvectors of the covariance matrix of the input images and the weight matrix.

5 Conclusion

Our implementation of the cortical memory dynamics produces positive results which support the analytical results of the net. In the four nodes (two per layer) case, by adjusting the Hebb constants, we were able to find the convergent state of the net which produces correct responses for input cues. We also tested alternative dynamics which have better convergence properties. However, the training process is difficult. Currently, we can only train a small net reliably.

References

[1] E. W. Kairiss and W. L. Miranker, "Cortical Memory Dynamics," Nov. 1995.

[2] Francesco Palmieri and Jie Zhu, "Learning in Linear Neural Networks," *IEEE Transactions on Neural Netowrks*, Vol.6, No.5, September 1995, pp 1165–1184.

[3] Geoffrey E. Hinton, Peter Dayan, Brendan J. Frey, Radford M. Neal, "The "Wake-Sleep" Algorithm for Unsupervised Neural Netowrks" *Science*, Vol.268, 26 May 1995.

[4] Rao and Dana Ballard, "Dynamic Model of Visual Memory Predicts Neural Response Properties In The visual Cortex," Technical Report 95.4 of University of Rochester, Nov. 1995.