Neural Net Applications

Willard L. Miranker
April 2009
TR-1416

Table of Contents

Filtering Spam Using Keywords and Features From Historical Data 1
Kwabena Agyei Antwi-Boasiako

Self Organizing Maps for Mesh Size Reduction 5
Steven Canfield

Artificial Neural Networks for Polyphonic Pitch Detection 9
Nathaniel Granor

Simultaneous Q-learning in Iterated Prisoner's Dilemma 13
Minh-Tam, Le, Frederick Shic

VLSI Design of Analog Neural Networks for Pattern Recognition 19
Dzmitry Maliuk

Using Neural Networks to Estimate Volatility of Financial Returns 25
Wedzerai V. Munyengwa

Neural Networks Playing Tic Tac Toe Trained by Backpropagation 29
Zachary Murez

Hopfield Network for Clustering 33
Huan Wang

Self-Organized Online Network Traffic Forecasting using Neural Networks 39
Ye Wang

Filtering Spam Using Keywords and Features from Historical Data

Kwabena Agyei Antwi-Boasiako

***Abstract*–A spam filter that uses an artificial neural network to classify email based on keywords is described. The algorithm and the parameters it uses are altered in order to optimize its performance. The filter is then tested on real data from two email accounts. The results of this project show that even though the spam filter relies on only a few characteristics of received messages for classification, it compares favorably with other methodologies currently in use on the market. The broader theme of identifying a document that is similar to another document is discussed.**

Index terms: email, historical data, neural network, spam filter

I. INTRODUCTION

The internet and email have become indispensable tools in our lives, mainly because of how convenient it is to relay messages that can be delivered almost instantaneously. Our dependency on this powerful email tool has afforded some the opportunity to profit from the exploitation of its vulnerabilities. In this project, spam is defined as unsolicited email that is sent from another user on the internet who often obtains the recipient's address illegally while keywords are words that appear in spam but do not typically appear in regular mail.

The experiment is broken into two parts. In the first part, the back-propagation algorithm [2] is used to train an artificial neural network to recognize keywords that are frequently used in spam emails to a particular email account. The aim of this classification is to investigate the correlation or similarity between words that appear frequently in spam. In the second part, the self organizing map algorithm [2] is used to classify email into two sets—"spam" and "regular," based on three characteristics: the length of the message, the fraction of words that are classified as "spam," and the fraction that are classified as "not spam." Finally, the two methods are compared and projections are made for how to optimize the filtering process.

The results showed that even with very little information about a given email account and its history, using artificial neural networks makes the process of filtering spam very efficient.

II. MATERIALS AND METHODS

All coding and simulations were done in C and Matlab.

A) Approach 1 (Training two-layer feed-forward network using the back-propagation algorithm)

Input to the neural net is generated from vectors containing the ASCII representation of strings that have been classified as either "spam" or "regular" [1]. The procedure used to generate keywords is outlined below:

Procedure:

1. Given an email account, randomly select a sample of messages that the user has classified as "regular" and "spam" respectively.
2. For each of the messages, generate all the strings it is made up of.
3. Generate the keywords for both "regular" and "spam", delete every word in the spam list that is also in the regular list.
4. Convert the characters of each string to its ASCII representation.
5. Generate input to the network by forming a random sample of words that have been classified as "spam" and "regular" respectively.

The desired output is defined as follows: output 1 if the word is "regular" and output 0 if it is "spam." The results of the simulation using the back-propagation algorithm are shown in Fig. 1 and Fig. 2.

The plot in Fig. 1 shows the correlation between words that appear frequently in spam. The regression factor is 0.56423, which shows that the correlation is not very strong. Fig. 2 on the other hand shows the performance of the trained neural network in classifying a set of inputs. The Class "regular" represents "regular" email while the Class "spam" represents "spam" email. The statistics in the confusion matrix [3] show a 74.1% accuracy in classifying email. It also shows that 3 "regular" emails (11.1% of total input) were incorrectly classified as "spam" while 4 "spam" email (14.8% of total input) were incorrectly classified as "regular."

B) Approach 2 (Classifying email into two classes based on 3 characteristics)

This approach also relies on the "dictionary" of

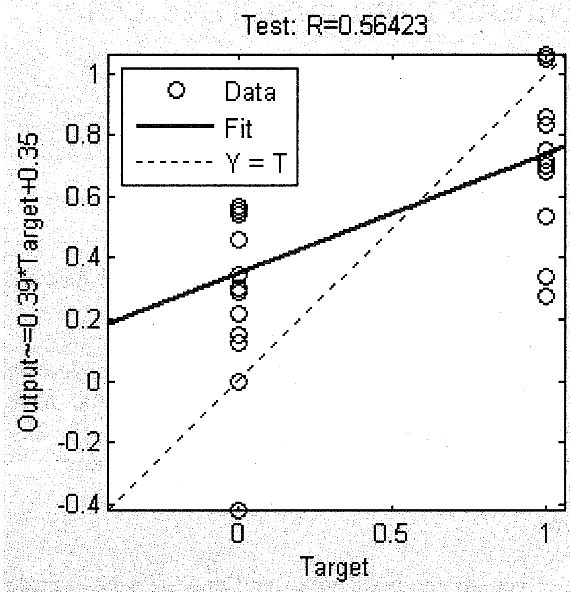


Fig. 1 Plot showing clusters of words on a dataset of 27 inputs

words generated as in Approach 1 except that the words are not converted to their ASCII equivalents. In this approach three characteristics of email are singled out:

i) The length of the message
ii) The fraction of the words it contains that are classified as spam
iii) The fraction of words it contains that are classified as regular

Output Class \ Target Class	regular	spam	
regular	13 48.1%	4 14.8%	76.5% 23.5%
spam	3 11.1%	7 25.9%	70.0% 30.0%
	81.3% 18.8%	63.6% 36.4%	74.1% 25.9%

Fig. 2 Map (confusion matrix) showing how neural network performed on test data of 27 inputs

Every message is represented by a vector with three components in the order listed above

$M = [x_1, x_2, x_3]^T$,
where M is the message,
x_1 is the length of M,
x_2 is the fraction of words in M classified as spam,
x_3 is the fraction of words in M classified as regular.

The inputs to the net are generated by randomly selecting messages from the user's email account and extracting these 3 features. The self organising map algorithm [2] is used to classify every message based on its representative vector. The outputs are shown in Fig. 3.

Fig. 3 shows the confusion matrix for 20 inputs. "regular" emails is classified as Class "regular" while "spam" email is classified as Class "spam". The results show that all 10 "regular" emails were correctly classified while 1 out of 10 "spam" emails was incorrectly classified. This gave the neural network an overall accuracy of 95%.

III. RESULTS

Fig. 1 and Fig. 2 show that the results of using Approach 1 are satisfactory given that the ASCII representation, which is basically a set of numbers (0 to 127 in this case), was used to represent strings. The

Output Class \ Target Class	regular	spam	
regular	10 50.0%	1 5.0%	90.9% 9.1%
spam	0 0.0%	9 45.0%	100% 0.0%
	100% 0.0%	90.0% 10.0%	95.0% 5.0%

Fig. 3 Map (confusion matrix) showing the distribution of 20 emails from inbox A

challenge in using this approach is that even though some words might look similar to the human reader, their ASCII representations will be totally different. For example,

"ROLEX" and "r.o.l.e.x" have ASCII representations "82 79 76 69 88" and "114 46 111 46 108 46 101 46 120" respectively. These make it difficult to classify messages based on keywords that have been slightly modified.

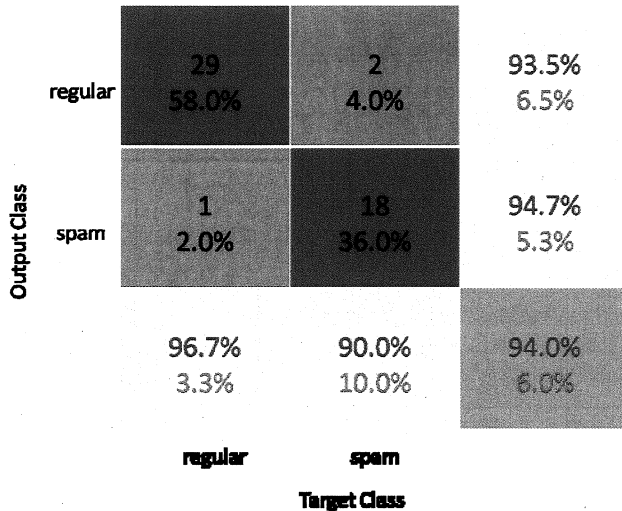


Fig. 4 Map (confusion matrix) showing distribution of 50 emails from inbox B

In spite of this, the weights showed a considerable amount of "closeness" which suggested a method that uses clustering of data to classify messages, hence the use of Approach 2. Observation of the results of Approach 2 confirms our guess about our data in Approach 1. The self organising map algorithm classifies the messages to a high degree of accuracy even employing only three somewhat crude parameters. Out of 20 messages (10 regular, 10 spam) that were tested, all the regular mails were correctly classified and only one spam message was misclassified (Fig. 3). Approach 2 was tested on emails from another email account and the results (see Fig. 4) were consistent with the one obtained in Fig. 3.

IV. CONCLUSION

The results show that neural networks are an effective tool for filtering spam based on the user's history. We can fine-tune even more message parameters in the user's email account, for example, it might be instructive to utilize other parameters such as the N most frequent words, the N least frequent words and so on. Also, to address the issue of false negatives (regular mail being classified as spam), it might be helpful to have access to the user's contacts list so that regular correspondents that the user certifies as genuine will have their messages always classified as "regular" even when the neural network suspects it is "spam." Augmenting the neural network with these parameters presents a very promising attack on the problem of filtering spam.

V. DISCUSSION

Even though the results obtained by using Approach 1 were less accurate than those obtained using Approach 2 (see Table 1), the weights exhibited an interesting property. There was a considerable amount of clustering even though the idea of "spam" and "regular" messages being similar to one another was not exploited. In recent times, the problem of finding a document similar to another document [4] has become a subject of interest mainly because of how web searches are made. More often than not, we tend to think of documents as being similar if they have the same author or they address the same subject.

	Approach 1	Approach 2
No. of inputs	27	20
% regular correctly classified	81.3	100
% regular misclassified	18.8	0
% spam correctly classified	63.6	90
% spam misclassified	36.4	10
Overall performance (% correctly classified)	74.1	95

Table 1 Comparison of the performance of the two approaches

This is not very different from thinking about these features as keywords with which we can generate parameters of every document and compare them based on only a few components using the power of neural networks.

VI. REFERENCES

[1] Clark, J., Koprinska, I., Poon, J. A neural network based approach to automated email classification. Proceedings, IEEE/WIC international conference on Web Intelligence, 2003.

[2]Haykin, Simon, Neural Networks, A Comprehensive foundation, Prentice Hall, Inc, 1999.

[3] http://en.wikipedia.org/wiki/Confusion_matrix

December 2008

[4] Saraçoğlu, R., Tütüncü, K., Allahverdi, N. A fuzzy clustering approach for finding similar documents using a novel similarity measure. Expert Systems with Applications, Volume 33, Issue 3, October 2007, Pages 600-605.

Self Organizing Maps for Mesh Size Reduction

Steven Canfield

Abstract—**In computer graphics, height maps are frequently used to represent three dimensional terrain. They are a compact representation of terrain data and can be easily edited by artists or an automated process. Unfortunately, naive height maps suffer from a number of problems, notably a poor mapping of triangle density to scene complexity. This paper explores adapting self organizing maps to provide a high quality mesh from height map data, and concludes that the self organizing map algorithm works well for this task when initialized with good starting data.**

Index Terms – **Self Organizing Map, Triangle Mesh, Height Map**

I. Introduction

In computer graphics, a *mesh* is a set of vertices, edges, and faces (Akenine-Möller, [1]). Each *vertex* has a position (x, y, z) in 3D space, and an *edge* is simply a connection between two vertices. A *face* is a collection of edges, most frequently a triangle with three vertices and three edges.

A *height map* is used to render landscapes or simple surfaces. Given source data of height values, a vertex is placed at every integer x, y with the z coordinate being the height at that point. The vertices are then connected to create faces to render the surface. Often, the source data is given in an image file where white represents the highest point and black represents the lowest point.

The self organizing map algorithm is frequently used to reduce the dimension of data, for clustering or viewing purposes. The task for this paper is to begin with some large amount of three dimensional data and reduce the size of the data while remaining in three dimensions. This paper introduces self organizing maps in Section II, height Maps in Section III, and the modifications to SOM in Section IV. Results are presented in Section V and Discussion in section VI.

II. Self Organizing Maps

A. Capabilities of self organizing map algorithm

The self organizing map algorithm is frequently used to analyze or display multidimensional data. This is accomplished by building a small dimensional (1,2, or 3) representation of the higher dimensional input. This representation preserves important topological features of the input data. Self organizing maps can also be used to approximate input data, which is the application we focus on here. For example, a SOM algorithm can take a two dimensional set of points and approximate it with fewer points (Haykin, [2]).

Yale University

B. Algorithm description

The Kohonen algorithm for self organizing maps has four steps (Kohonen, [3]).

1. Select an input vector x at random.
2. Find the neuron y that is nearest (Euclidean distance) to x.
3. Calculate the neighborhood $N(y)$.
4. Update the weights of each neuron in $N(y)$ by $\eta(x - y)$, where η is an arbitrary learning rate.

III. Height Maps

A height map is a two dimensional array M where $M[x, y] \in [0, 1]$ is the height at (x,y). Most frequently, triangles are used to connect each (x,y) point to form the surface of the rendered height map, as shown in Figure 1.

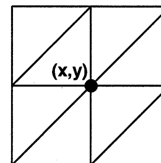


Fig. 1. Triangle faces that contain (x,y)

Traditionally this array is loaded from an image file where one of the color channels is interpreted as the height. Figure 2 shows an example height map stored as an image file.

This is a compact representation since only the height value must be stored (the x and y coordinates are implicit in the order of the data). Unfortunately, since the points are evenly spaced on a grid, height maps suffer from a few shortcomings.

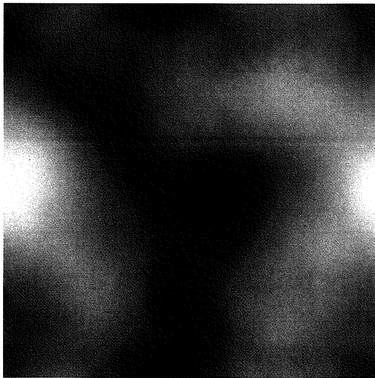


Fig. 2. Computer generated height map

A. *Height Map Problems*

The salient limitation of a standard height map process is that it does a poor job of distributing vertices, since it distributes them evenly over complex and simple areas of the map. For example, if one region of the map has a constant slope, it can be represented with a very small number of triangles. In contrast, if the slope changes rapidly, as many triangles as needed to accurately represent the terrain should be used.

A secondary problem relates to texture mapping. A texture is a bitmap image to be overlaid across a mesh; it defines the color of each point on the mesh. In most texturing models, a texture coordinate pair (s,t) is assigned for each vertex. When a face containing the vertex is rendered, the texture is interpolated across the face. This means that if there are more pixels on screen than between the given texture coordinates, some colors from the texture will have to be repeated or interpolated. Thus, to texture a height map, we want to generate a set of texture coordinates to be used at each vertex so that the texture is spread across the height map. Minimizing the size of the largest face will ensure that the texture is displayed as well as possible.

Figure 3 shows both of these problems in action.

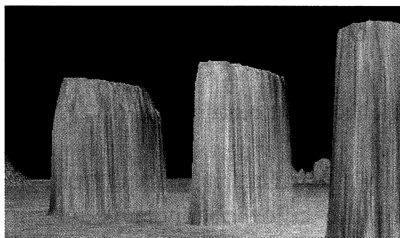


Fig. 3. Poor vertex distribution and texture mapping

IV. Self Organizing *Mesh*

The self organizing mesh algorithm takes as input a matrix of height values and returns a list of (x, y, z) tuples for use as a mesh. This representation is three times larger than the input since it must now explicitly store the full tuple. Each node in the network has a weight vector of size 3 representing the (x, y, z) coordinates for the point. Weight vectors are initialized with the given $(x, y, 0)$ in contrast to the generic SOM algorithm. Finally, there are additional constraints on the map algorithm. Where in the original self organizing map algorithm, each node could move freely, in our algorithm the nodes are confined in the sense that they must stay within the grid. So if we move an element west, we must not move it further than the x coordinate of its neighbor nodes or we would break the mesh. This is important because generating the mesh from a set of x, y, z points is extremely difficult, but keeping our simple mesh from the beginning of the process is simple.

An addition to the self organizing map algorithm called the "conscience" algorithm is also employed here. In the "conscience" algorithm, a winning neuron is penalized in future rounds to allow other winners. The penalty decays with each round which allows that neuron to eventually win again.

The pseudocode for the self organizing mesh algorithm (without conscience) is shown below.

A. *Algorithm*

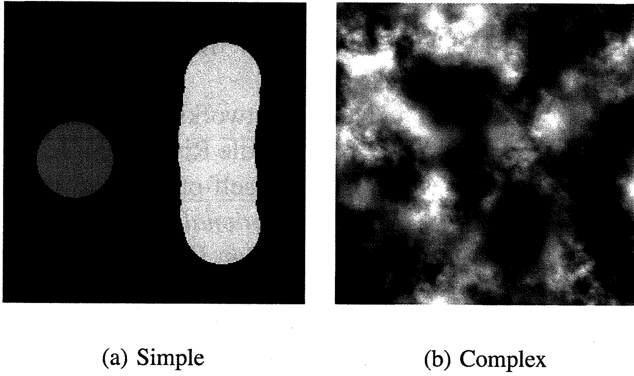
```
function SOM( H )
    for x in 0 to H.width
        for y in 0 to H.height
            rx = random(-0.5, 0.5)
            ry = random(-0.5, 0.5)
            M.append(x+rx ,y+ry ,0)

    numIter = 0
    while(numIter < MAX_ITERATIONS)
        numIter++
        height = H[Rx][Ry]
        node = closestPoint(Rx,Ry,height)
        neighborhood = getNeighborhood(node, numIter)
        for each node in neighborhood
            move (Rx,Ry,height)
        end
    end
end
```

V. Results and Methodology

We used two principal data sets for testing. The first is a simple test with two main features, a round hill on the left and a taller ridge on the right. The second data set is one generated using terrain generation software and is representative of real world three dimensional maps. Figure V shows both image input files.

The self organizing map algorithm works well at generating smaller meshes, and shows gains in improving the quality of a mesh. The output still has some of the problems of a naive height map but they are reduced. For all of these tests a 256×256 vertex input was reduced to a 128×128 vertex mesh.



(a) Simple (b) Complex

Fig. 4. Image files for SIMPLE (a) and COMPLEX (b)

The simplest image in the data set is SIMPLE, which can be seen in Figure 4(a). There are two features in this mesh, a low mass on the left and a longer, higher mesh on the right. The result of the SOM algorithm (with 5000 iterations) on SIMPLE can be seen in Figure 5(a). The flat terrain is somewhat improved, as can be seen in a close up of the lower left corner in Figure 5(b). The sparseness of flat terrain has also allowed more vertices in the very steep region, as can be seen in the side view in Figure 5(c).

The more interesting image in the data set is COMPLEX, which can be seen in Figure 4(b). Figure 6(a) shows a rendering of COMPLEX at full resolution (that is, using normal height mapping). Figure 6(b) is the output from the SOM at 2000 iterations, since the COMPLEX data set had much better results with fewer iterations than the SIMPLE set.

Finally, we utilized the "conscience" addition to the SOM algorithm to generate Figure 7.

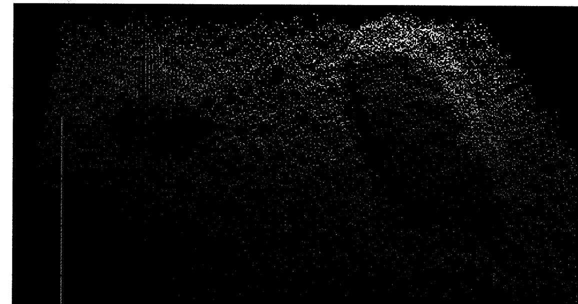
VI. DISCUSSION

There are a number of limitations to the SOM algorithm when used for mesh size reduction. The first is the number of iterations required, especially as the size of input increases. Although in theory this algorithm would be part of an automated process, a human would have to inspect the results since the algorithm may produce poor samples depending on the random choices made. A second limitation of this algorithm is that there are many parameters that must be adjusted to achieve good results. For example, the algorithm's user must tune the neighborhood function, the decay of its inputs, the number of iterations, and the strength of each movement. The "conscience" algorithm performs better than the standard algorithm so future research should begin there.

Future research into this method would begin with developing a metric for quality of mesh representation.

(a) Simple: Output

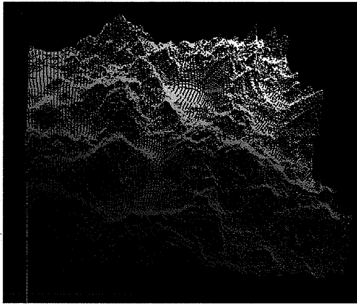
(b) Simple: Sparse Mesh in Flat Region



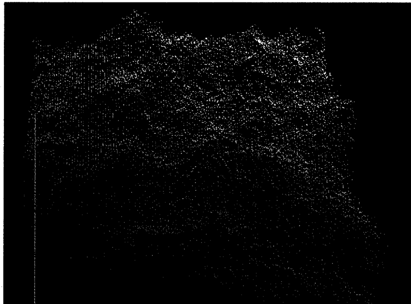
(c) Simple: More Detail in Steep Region

Fig. 5. Results for SIMPLE. Note the improvements in mesh quality in (b) and (c), as well as the overall representation.

For example, one simple metric might be the area of the largest polygon (which should be minimized). A second area to explore would be to decrease the complexity of the algorithm as given so that a closer representation could be achieved. For example, we found good results with small ($64 \times 64 \rightarrow 32 \times 32$) images and a very small neighborhood function, but the running time would make that strategy impractical. Finally, we would consider a functional description of the landscape as the input to this process. For example, consider defining the landscape as



(a) Complex Full-Scale Render



(b) Complex SOM Output

Fig. 6. Results for COMPLEX. The full resolution render is shown in (a) and the SOM output is shown in (b)

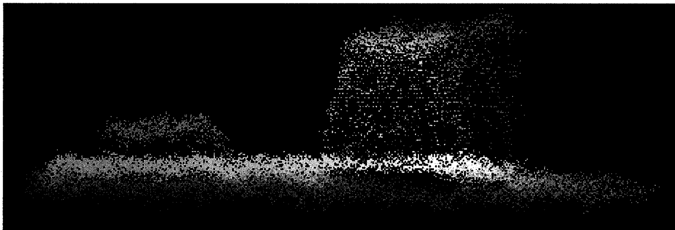


Fig. 7. Simple output with conscience algorithm

the curve formed by $z = \log y^{\sin x}$. Extrapolating from the success of two dimensional maps at representing two dimensional functions leads us to hope for good results in three dimensions.

VII. BIBLIOGRAPHY

1. Akenine-Möller, T. et al, 2008. "Real-Time Rendering 3rd Edition", Natick, MA.

2. Haykin, S., 1999. "Neural Networks," *Self Organizing Maps*, pp. 480-481, Upper Saddle River, New Jersey.

3. Kohonen, T., 1990. "The self-organizing map," *Proceedings of the IEEE International Conference on neural networks*, pp. 1147-1156, San Francisco.

Artificial Neural Networks for Polyphonic Pitch Detection

Nathaniel Granor

Abstract— **In this project, the application of an Artificial Neural Network with Backwards Propagation Learning to the problem of Polyphonic Pitch Detection in sampled audio is analyzed. This procedure would be one possible step in a larger signal processing application for the transcription of music from digital audio files. Backprop is found not to be a satisfactory pattern matching algorithm for this problem.**

Index Terms— **polyphonic pitch detection, neural networks, backwards propagation, constant Q**

I. Introduction

Music is a human abstraction of sound. We talk about music in terms of distinct "notes," which are related to pitch or the frequency of a sound wave. But when music is played on instruments (including the human voice), many different frequencies of sound are actually generated. These frequencies are the fundamental pitch and a theoretically infinite sequence of overtone, which musicians call the "overtone series" or "harmonic series". Most musical instruments generate an arithmetic sequence of overtones; sound waves are produced at integer multiples of the fundamental frequency. A "note" in a piece of music indicates, by musical convention, which fundamental pitch should be produced. The overtone series that is also produced is determined by the acoustic properties of the instrument that plays it. No two instruments can be assumed to produce the exact same series of overtones. [1]

Most humans are able to pick out the fundamental pitches in the sounds they are interpreting as music, regardless of the set of instruments involved. Most of the time, the brain is able to process data indicating which frequencies are present and at what amplitudes, and detect which of the frequencies represent musically relevant pitches and which are artefacts of the acoustic properties of the instrument. [2]

But this process, and its derivatives, such as manual transcription of music, is time-consuming and mentally taxing. A well-trained musician would require multiple listenings to transcribe all of the notes present in a piece of music. It is desirable to create software capable of perceiving music from sampled audio input (for example, to produce a transcript of a musical performances). In practice, detection of pitch in monophonic music (one note played at a time) is, relatively trivial, since the fundamental pitch is most likely the frequency resonating at the highest amplitude. Detection of pitch in polyphonic music, however, is much more difficult, particularly if, as is typically the case, the number of notes present and the properties of the instruments producing them are unknown.

A procedure for producing a musical transcript from a digitized audio recording might be as follows: Break up the recording into "frames". A new frame begins each time a new musical note is sounded or dissipates. Determine which fundamental pitches are playing in each frame. Neither of these steps is elementary. This project focuses on the second step.

II. Fourier Analysis

Fourier Analysis is well suited to transforming a composite waveform (for instance, the sum of all the sound waves present) into its constituent parts (sound waves at each of the frequencies present). A Fast Fourier transform is capable of doing this job very efficiently.

It would be inefficient and undesirable if the Fourier transform produced values for every possible frequency (i.e. for every component in its infinite series of oscillatory functions). We can instead produce a histogram by selecting a finite number of points in time and the distance apart to space them. The resulting output is a series of frequency bins or bands and the net amplitude of frequencies in each band.

However, the musical abstraction of pitch from sound is on a logarithmic scale. Low musical notes are separated by significantly smaller differences of frequency than high notes. The Fourier transform produces output in linearly spaced bins. In the musical domain, this results in differing levels of frequency resolution in different musical ranges.

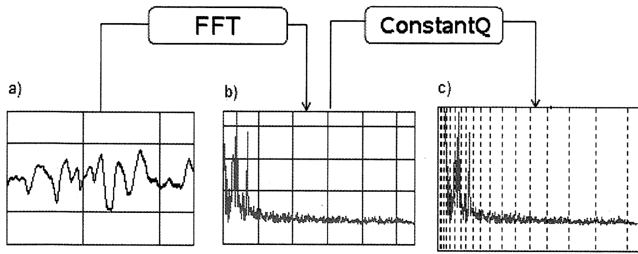
III. Constant Q Transform

Judith Brown, 1990 , describes an elegant modification to the Fourier transform, called the Constant Q transform, that uses geometrically spaced center-frequencies for the bins. Like human ears, the Constant Q transform has increasing time resolution towards higher frequencies, and trades time resolution for frequency resolution to identify low-tone frequencies. [3] Fig. 1, below, illustrates the differences in bin assignment between a Fourier transform and the Constant Q transform. Brown later published an efficient algorithm for calculating the Constant Q transform [4] and it is Benjamin Blankertz' Matlab implementation of this algorithm that we use to perform our spectral analysis [5].

IV. Distinction Between Pitch and Pitch Class

In the musical domain, each separate note has a precise pitch that belongs to one of 12 pitch classes. By convention,

Fig. 1: Comparison of Constant Q and Fast Fourier transformations from [8]



the pitch known as "middle A" has a frequency of 440Hz. It's pitch class is called "A". The twelve pitch classes are arranged serially and repeat. Each repetition (up or down) of the 12 pitch classes is called an octave. Two pitches an octave apart have a 2:1 frequency ratio. So, the note precisely one octave above "middle A" has a frequency of 880Hz. It is also placed in the pitch class "A".

For harmonic analysis applications (Identifying the underlying chord, determining major or minor tonality, etc), pitch class is more important than pitch. The focus of our application is not pitch class, but rather pitch itself.

V. Pattern Matching with Artificial Neural Networks

Preprocessing each frame of sound using the Constant Q Transform produces a data vector of n components, where n is the number of bins specified. The number of bins corresponds to the number of discrete notes allowed in our analysis. The data contained in this vector do not indicate what the fundamental pitches are (which musical notes were played). Rather, the data indicates how much vibration is happening within the frequency bands surrounding each musical note. Because instruments typically produce overtones only at integer multiples of the fundamental pitch, we may assume that each frequency detected is either a fundamental frequency of a note being played, or an integer multiple of such a frequency. This restriction simplifies the problem sufficiently that it may be possible to extract the original fundamental pitches given the output of a Constant Q transformation. This extraction requires some sort of pattern matching algorithm, and an Artificial Neural Network will be used.

An Artificial Neural network is composed of model neurons arranged in a finite number of interconnected layers. Each neuron takes multiple inputs (supplied by other neurons) and produces an output from a discriminating function acting on the weighted sum of the inputs. In effect, each connection between neurons (or between neurons and input) has a weight. A neural network can be trained on a set of training data using a learning rule. [6]

Our neural network will take n inputs (where n is the number of bins in the Constant Q transform) and produce n binary outputs, each of which indicates whether a particular pitch is a fundamental pitch present in the current frame. The internal structure of the network and the learning rule will be the subject of our investigation.

Backward Propagation

We use *supervised learning*, where the neural network first goes through a directed training phase. During training, the network weights are initialized (perhaps randomly) and then the network is trained on input/output pairs. The network computes the difference between the expected output and the output actually produced and updates its weights based on that error using the Backprop algorithm. This training step is repeated until the weights converge to stable values.

Back-Propagation learning calculates a local gradient for each neuron in the network, moving backwards layer-by-layer. In this sense, the network goes through a forward and backward pass for each training I/O pair: The input values are presented and the neurons propagate their output values forwards, layer-by-layer; Then, the error is calculated and the local gradients are fed backwards, allowing each neuron to calculate its new weight. [6]

VI. Production of Test Data

Paired input/output data are required to test the pattern matching and to train the network. We are faced with a dilemma: producing such test data would require manually transcribing audio files, which is precisely the difficult and time-consuming activity we wish to avoid. Lee and Slaney proposed a clever workaround for this situation [7]. The MIDI file format is a computer file type that encodes songs in terms of their musical components. MIDI files consist of "events" such as "NOTE ON" and "NOTE OFF" which specify what notes get played when. If we start with a MIDI file, we can easily produce a matching digital audio file. Careful manipulation of the original MIDI file can tell us where the frames are in the audio file and what notes are playing in each frame. Figure 2, below, illustrates the training process.

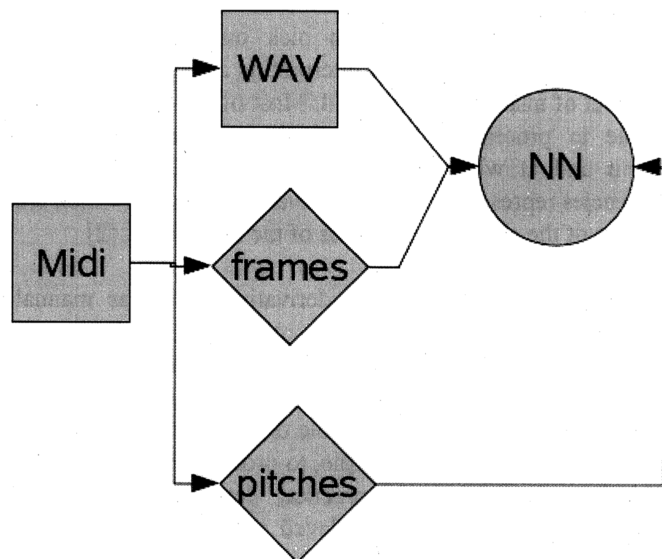


Fig. 2 illustrating training preprocess

The pitch and timing data from the MIDI file are separated. The timing data are used to calculate the frames, while the pitch data are used as the expected output.

MIDI files, however, contain only musical data. The contents of our digital audio file will depend upon the MIDI player that creates the audio file. MIDI players use a variety of sound synthesis algorithms that crudely estimate real instruments. Human voices are even more difficult to synthesize, as the sound properties vary based on the changing shape of the singer's mouth and throat. So our test data (and training data) will probably be simpler than real audio data. The resulting pattern-matching will probably work better for our test data than for real audio samples.

Our training sets and sample data were taken from a simplified problem space: we used only one MIDI instrument (Grand Piano) in generating all of the audio data.

METHOD FOR EVAULATION

Using the same method which produced the training data, we can produce test data. The nature of this data is that the vast majority of values in each output vector should be zero. Instead of taking a mean value of the error, we calculate the proportion of false positives (occasions where the network indicated a pitch is present and it shouldn't be) to total pitches and the proportion of false negatives (occasions where the network missed a pitch that was present) to total non-present pitches.

MONOPHONIC TESTS

Monophonic pitch detection should be a simple case of polyphonic pitch detection. So the first training set consists of the 88 chromatic pitches played slowly in sequence.

We begin with a simple net with no hidden layers. The network simply maps the 88 input values to the 88 output values. The network converged quickly with high performance on the training data. However, the trained network performed poorly on the first test series (Andersen Flute Etude Op 42, Num 12).

This result came as a surprise, since monophonic pitch correction should be pretty straightforward. However, test 1 consisted of fast-moving notes and the training set consisted of slow-moving notes. The onset of a musical note, called the attack, contains a lot of extra noise that quickly dissipates. This first test contains much more "attack noise" than the training data, and this could account for the poor performance.

A second test was performed with a monophonic melody comparable in speed to the training data (melody from "You'll Never Walk Alone"). This test showed a slight improvement.

Another possible source of error is reverberation (echo) in the digital audio files. Extraneous frequencies might be bleeding in between frames. This is a more serious problem for the training data than for the test data.

We add a hidden layer of 166 neurons (twice the size of the input and output vectors) hoping to improve the result. This number of neurons in the hidden layer was selected after observing that a larger number of neurons led to sharp increases in calculation time for the network with constant efficiency in the performance of the network, while smaller numbers of neurons had lower efficiency.

Repeating the same monophonic tests on this new network delivered only slightly better performance. See table 1.

	Single layer (88)		Double layer (166,88)	
	% false negatives	% false positives	% false negatives	% false positives
Training data	0	0	0	0
Test1 (fast)	94.92	1.14	90.85	1.15
Test2 (slow)	50.63	0.95	50.63	4.79

Table 1 showing monophonic results in two different networks

We speculate that the simplicity of the monophonic pitch detection problem might be clouding the results. In the more complex polyphonic examples, there will be considerably more noise in the spectral data and perhaps the additional noise will help the network make significant determinations.

POLYPHONIC TESTS

We used several different training sets made up for combinations of the following: major and minor chord triads in each octave, chromatic scales, random cluster chords and random well-spaced chords.

We tested the two monophonic songs as well as an additional polyphonic song ("Imagine" by the Beatles). See table 2.

	Major/Minor triads (166,88)	
	% false negatives	% false positives
Training data	0	0
chromatics	7.35	6.31
Test3 (poly)	93.96	3.26

	Triads and chromatics (166,88)		Triads, chromatics, random chords (166,88)	
	% false negatives	% false positives	% false negatives	% false positives
Training data	0	0	19.96	0.003
Test1 (mono)	94.11	0.92	94.31	1.73
Test2 (mono)	39.24	7.41	39.24	7.33
Test3 (poly)	99.31	0.22	78.72	7.13

Table 2.Results using polyphonic training sets

VII. CONCLUSION

The marginal accuracy of the results generated suggests that a feed-forward net with backpropagation is not an adequate tool for the pattern matching component of polyphonic pitch recognition.

That most of the network configurations tested converge efficiently and perform flawlessly on their training sets defends the idea that a neural net might be able to fulfill this function. The results might benefit from the following adjustments to the procedure: expanded training data sets, a fancier preprocess that can better account for real effects like reverb, and a small, fixed frame size.

Another logical extension would be to replace the Back-Propagation network with another form of pattern matching tool. A self-organizing map seems well suited to the task. Further work is required to determine how the problem might be restated in a domain suitable for a self-organized map.

REFERENCES

[1] R. Bain, "A Web-based Multimedia Approach to the Harmonic Series," Available online: http://www.music.sc.edu/fs/bain/atmi02/.

[2] "Psychoacoustics," Wikipedia: the free encyclopedia. Available online: http://en.wikipedia.org/wiki/Psychoacoustics.

[3] J. C. Brown, "Calculation of a constant Q spectral transform," *J. Acoust. Soc. Am.,* vol. 9, no. 1, pp. 425-434, January 1991.

[4] J. C. Brown and M. S. Puckette, *J. Acoust. Soc. Am.*, vol. 92, no. 5: pp. 2698-2701, 1992..

[5] B Blankertz, "The ConstantQ Transform" *Draft.* Available online: http://ida.first.fhg.de/publications/drafts/Bla_constQ.pdf.

[6] S. Haykin, *Neural Networks, Second Edition*. New Jersey, Princeton Hall Press, 1999.

[7] K. Lee and M. Slaney, "Automatic Chord Recognition from Audio Using an HMM with Superised Learning," *Proceedings of the 1st ACM workshop on Audio and music computing multimedia,* pp. 11-20, 2006.

[8] G. Cabral, J. P. Briot, F. Pachet, "Impact of Distance in Pitch Class Profile Computation," *Sony Computer Science Lab,* Paris. Available online: http://gsd.ime.usp.br/sbcm/2005/papers/short-13848.html.

Simultaneous Q-learning in Iterated Prisoner's Dilemma

Minh-Tam Le, Frederick Shic

***Abstract*—The performance of the Q-Learning algorithm in the context of a multi-agent Iterated Prisoner's Dilemma [1] is assessed. The experiment shows that as the agent's memory size increases, the state space of the learning task also increases exponentially, thus preventing the agent to come up with an optimal solution for the task. A new algorithm, Simultaneous Q-learning is proposed as a variation to the basic Q-Learning algorithm [2] to deal with more complex state spaces. Instead of updating only the current state in each step, the new algorithm simultaneously updates the estimates of all states related to the current state. Experimental results show that an agent using Simultaneous Q-Learning learns more effectively in complex state spaces than an agent using Basic Q-learning.**

***Index Terms*—Reinforcement learning, Q-Learning, Iterated Prisoner's Dilemma.**

I. Introduction

Reinforcement learning (RL) refers to a family of learning algorithms which rely on the environment's feedbacks on committed actions to guide future actions. RL is applicable where the learning agent is not supplied with a dedicated set of training examples, but must instead explore the environment by taking actions and receiving evaluations from the environment. The agent's goal is to learn to select actions which maximize utility. The agent learns by taking exploratory actions, receiving feedbacks and updating its utility estimates. In RL's learning process, in a particular situation (state), the tendency to take an action which yields favorable outcomes should be reinforced, while an action which produces unfavorable results should be discouraged.

Q-learning [2] is an RL algorithm, which learns by incrementally estimating the expected values of state-action pairs. Many algorithms have been proposed for learning sequential tasks in single-agent context using reinforcement learning in general and Q-learning in particular [3-5]. Q-learning has been applied particularly in robotics [6,7], and game-playing and other multi-agent tasks [8,9].

Let $Q(s,a)$ denotes the expected value of the discounted sum of future gains after taking action a from state s and subsequently undertaking an optimal policy. Assume that if an agent takes an action a from a state s, it will arrive at a next state s', after receiving an immediate payoff r. The Q-learning algorithm learns from this experience, and correspondingly updates $Q(s,a)$:

$$\Delta Q(s,a)=\alpha[r+\gamma \max_b Q(s',b)-Q(s,a)] \quad (1)$$

where α is the learning rate and $0 \le \gamma < 1$ is the discount factor. The activity is carried out until the algorithm converges to the optimum action-values. At the end of the learning process, the optimal action from any state is the one which corresponds to the highest Q-value.

The Q-learning algorithm is guaranteed to always converge to the optimum, given that all actions are sampled infinitely often from all states [10]. However, in practice, the large size of the state space, combined with that of the action set, renders exhaustive exploration of all state-action pair unfeasible. Therefore, when its memory size increases, the performance of the learning agent suffers as the state space complexity increases. The challenge is thus for the agent to *generalize from a relatively limited number of explored state-action pairs to induce the outcomes of many other unknown, unexplored pairs* [11]. Several solutions have been proposed: generalization between similar states using weighted Hamming distance and statistical clustering [6], or using self-organizing maps (SOM) [5] to capture the desired state-action mapping.

In this paper, we proposed a solution to deal with complex state spaces in the Q-learning task, making use of a neighborhood function (like that of SOM) to generalize similar states. The proposed method doesn't build an SOM, and hence differs from the existing SOM-based methods. The new approach is studied and tested in the context of the Iterated Prisoner's Dilemma problem.

II. Iterated prisoner's dilemma

The Prisoner's Dilemma (PD), a problem of game theory, refers to a social situation where each agent has two possible actions: to cooperate or to defect. A payoff matrix detailed in TABLE I describes a PD game if $T > R > P > S$.

TABLE I
Payoff matrix for the agent

		Opponent	
		Cooperate (C)	Defect (D)
Agent	Cooperate (C)	R	S
	Defect (D)	T	P

In addition, if the game is repeatedly played, thus considered an Iterated Prisoner's Dilemma (IPD) game, the inequality $2R > T + S > 2P$ must also hold [13]. These two inequality constraints effectively form a situation in which it is more beneficial for an individual agent to defect, regardless of the opponent's choice, although the sum of the agents' profits is maximized if both cooperate and minimized if both defect, hence creating a dilemma.

In an IPD game, a stochastic agent chooses its action from a distribution, which is a mapping from the entire history (of the agent's and its opponent's moves) to the utility values of the actions. It is also assumed that the two agents are not aware of how long the game lasts (in terms of iterations.) Then the goal of a learning agent in an IPD game is to maximize its discounted return

Digital Object Identifier inserted by IEEE

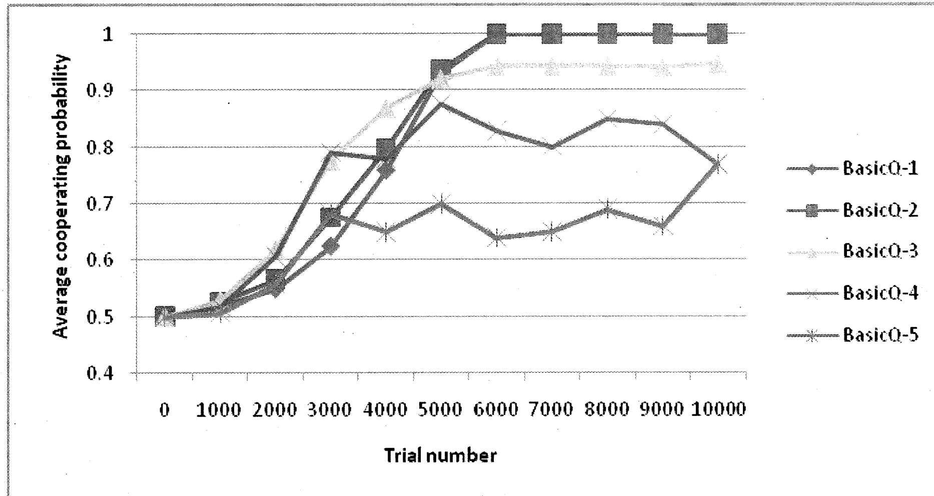


Fig. 1. Basic Q-learning algorithm performance in IPD game with increasing memory size.

$$\sum_{i=n}^{\infty} \gamma^{i-n} r_i \qquad (2)$$

where $0 \leq \gamma < 1$ is the discount factor and r_i is the immediate gain on the ith iteration. In effect, the discount factor influences the agent's consideration between the long-term cooperative incentive and the immediate gains for defection.

We are interested in training such an agent to play optimally against an opponent who employs a fixed strategy named Tit-For-Tat (TFT)[1] [1]. In his landmark study [14], Axelrod found that the optimal ways to play against a TFT opponent may differ, depending on the value of the discount factor γ. This result is summarized in TABLE II.

TABLE II

OPTIMAL STRATEGIES AGAINST TFT OPPONENT

$\gamma \geq 2/3$	Always cooperate
$1/4 \leq \gamma < 2/3$	Alternate between cooperation and defection
$\gamma < 1/4$	Always defect

III. PROBLEM WITH BASIC Q-LEARNING

Due to the cited computational limitations, the learning algorithm will employ a fixed number L ($L \geq 1$) of previous moves as the context to decide the next action. Thus, a "state" is defined to be, instead of the entire history of the game, a window of L previous moves. Each move may take a value from the set {CC, CD, DC, DD}, giving a total of 4^L different states. For each state, two Q-values are to be stored (for cooperating and defecting actions, correspondingly). Moves further than L-move away are ignored by the agent. The set of Q-values are therefore stored in a lookup table, called Q-table.

In each trial, the *Basic Q-learning* algorithm, as summarized in the Introduction section, identifies the state s in the Q-table which matches its memory and the state's corresponding Q-values, and then computes the agent's probability of selecting either action C or D [1]. However, after receiving its opponent's move and getting the immediate payoff, the agent updates the Q-values of only one state s. As the number of possible states grows exponentially with L, a single-state-update on every turn proves to be insufficient in "exploring all actions in all states infinitely often" as dictated by the convergence condition [10].

Fig. 1 plots the agent's average cooperating probability in each trial throughout the game. The algorithm was run with learning rate $\alpha = 0.2$, discount factor $\gamma = 0.95$, and the payoff matrix parameters $T = 0.5$, $R = 0.3$, $P = 0.1$, $S = 0$. The settings are taken from the work of [1]. According to , since $\gamma = 0.95 > 2/3$, we expect the cooperating probability of the agent to converge toward 1.0 at the end of the learning process.

There is significant decline in steady-state cooperating rate in Fig. 1 as the window size L is increased from 1 to 5, with the *BasicQ-5* line settles just about 0.77. With larger values of L, the cooperating rate could drop to about 0.5, which is just as good as a random player. This experiment shows that the learning agent suffers from the increasingly complex state space, which is the result of the agent's enlarging memory size (curse of dimensionality).

IV. SIMULTANEOUS Q-LEARNING

To deal with more complex state spaces, reaching the same or comparable performance as it gives to a small, simple state space, the agent must be able to reduce or generalize the complex state spaces. Various attempts [5,6,11] have been made to address this issue by proposing different ways of generalization of the state space, thus enabling the agent to simultaneously update multiple states in each trial. In line with these ideas, we proposed a weighted similarity measure to generalize the state space in the IPD game, and use that similarity value in updating other states which are similar to the current states.

Let a state S_i be denoted by a sequence of moves $[S_{i1}, .., S_{iL}]$ where S_{ik} is a 2-bit representation: $S_{ik} \in$ {CC, CD, DC, DD}, $1 \leq k \leq L$. S_{i1} denotes the most recent move between the agent and its opponent, S_{i2} the second most recent, etc. Then the similarity between states S_i and S_j is defined as:

[1] An agent using TFT initially cooperate on its first move and then keeps repeating its opponent's previous move (thus "punishing" the opponent's defection and "rewarding" cooperation on a one-on-one basis).

Fig. 2. Performance comparison between Basic Q-learning and Simultaneous Q-learning (A).

$$sim(S_i, S_j) = \frac{\sum_{t=1}^{L} \tau^{t-1}(S_{it} \otimes S_{jt})}{\sum_{t=1}^{L} \tau^{t-1}} \quad (3)$$

where $x \otimes y = 0$ if $x \neq y$ and 1 if $x = y$ (with $x, y \in$ {CC, CD, DC, DD}) and $0 < \tau \leq 1$ is the memory decay factor.

The parameter τ underlies an a priori assumption that the more recent past action is more important in deciding the future action, which is reasonable in the context of an IPD game. The states in the Q-table are updated, using:

$$\Delta Q(s,a) = \alpha \, \mathrm{sim}(s, s_{current})[r + \gamma \max_b Q(s',b) - Q(s,a)] \quad (4)$$

where $s_{current}$ denotes the current state.

V. Experimental results

A. Basic Q-Learning vs. Simultaneous Q-Learning (A)

With parameter values replicated from [1] and $\tau = 0.2$, we carried out the experiments to compare the performance of *Basic Q-learning* and the *Simultaneous Q-learning (A)*[2] proposed in the previous section. Fig. 2 shows the comparison between the cooperating-rate of the two algorithms, as memory size increases from 1 to 5.

For the cases where window size L = 1, 2, 3, the proposed Simultaneous Q-learning (A) cannot outperform the Basic Q-learning algorithm. However, there are two noteworthy points:

- Except for the case where $L = 1$, all the curves of cooperating rate performed by *Simultaneous-learning (A)* initially descends below 0.5 before jumping steeply upward, reaching their steady states. Although their steady states remain below their corresponding counterparts given by *Basic Q-learning*, it seems that the Simultaneous algorithm explores the defecting option to a greater extent before converging toward cooperating.
- With $L = 4$ and $L = 5$, *Simultaneous Q-learning (A)* clearly outperforms *Basic Q-learning*.

B. Basic Q-Learning vs. Simultaneous Q-Learning (B)

Although the performance of *Simultaneous Q-learning (A)* yields some interesting patterns, we were not satisfied with its asymptotic performance. The fact that it drops far below 0.5, "wasting time slowly exploring" alternative options, suggests that *Simultaneous Q-learning (A)* over-generalizes the states to a certain extent, i.e. it updates states which should not be deemed as similar to the current memory.

In an attempt to rectify this behavior, we introduce *Simultaneous Q-learning (B)* which contains a slight variation in the similarity measure computation in (2):

$$sim(S_i, S_j) = \frac{\sum_{t=1}^{K-1} \tau^{t-1}(S_{it} \otimes S_{jt})}{\sum_{t=1}^{L} \tau^{t-1}} \quad (5)$$

where K is the smallest index number such that $S_{iK} \neq S_{jK}$.

This modification, instead of including all memory bits of the two states in similarity computation, breaks off the computation immediately when the states' memory bits start to differ. The results of *Basic Q-learning* vs. *Simultaneous Q-learning (B)* are given in Fig. 3, which shows that the modified algorithm outperforms *Basic Q-learning* for every window size L from 1 to 5. However, the difference is still not very significant in the cases of L = 4, 5. Further refinement of the similarity function is required.

C. Investigation on the effect of parameter τ

We carried out one more experiment to investigate the effect of the memory decaying factor τ which is used in computing the similarity measure in the state space. In this experiment, we used the modified Simultaneous Q-learning (B) algorithm, with $\tau = 0.2$ and $\tau = 0.1$. All other parameters are identical. The result in Fig. 4 shows that the learner with $\tau = 0.2$ consistently outperform its competitor with $\tau = 0.01$.

[2] Since we are going to introduce some variation later on, for clarification purpose, we hereby call the algorithm originally proposed in the last section *Simultaneous Q-learning (A)*

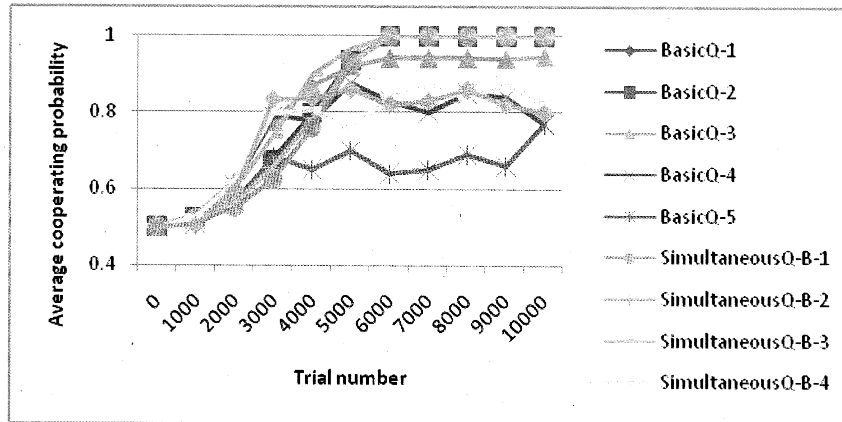


Fig. 3. Performance comparison between Basic Q-learning and Simultaneous Q-learning (B).

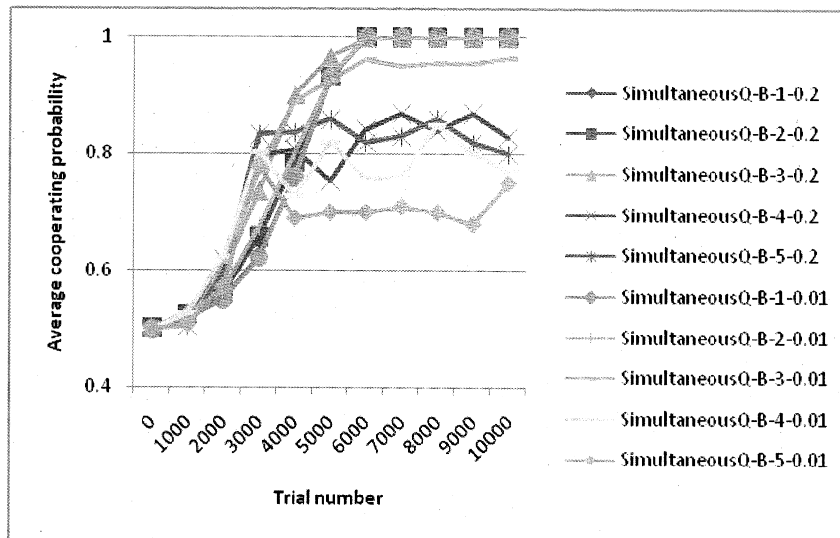


Fig. 4. Simultaneous Q-learning (B) with $\tau = 0.2$ and $\tau = 0.01$.

VI. Conclusion

We have shown, in the context of multi-agent IPD, the increasing size of the agent's memory results in the exponential growth of the state space and complicates the learning task. We have also presented Simultaneous Q-learning as a modification of the Basic Q-learning algorithm. The proposed algorithm relies on a similarity measure to determine neighborhoods of states, which enable state generalization and thus improve the learning effort. Simultaneous Q-learning is shown to perform better than Basic Q-learning in learning the optimal strategy against a TFT-agent in IPD. *Simultaneous multiple state-action pair updates allows more state-action pairs to be explored during the limited time of the learning process, thus enhancing the probability of convergence.*

Future work will involve refining the similarity measure, investigating the performance of Basic Q-learning and Simultaneous Q-learning agents against each other in an IPD game.

References

[1] T. W. Sandholm and R. H. Crites, "Multiagent reinforcement learning in the iterated Prisoner's Dilemma" in *BioSystems*, vol. 37(1-2). Elsevier Science Ireland Ltd., 1996, pp. 147-166.

[2] C. Watkins, *Learning from delayed rewards,* Ph.D. dissertation, King's College, University of Cambridge, UK, 1989.

[3] R. S. Sutton, "Learning to predict by methods of temporal differences," in *Machine Learning,* vol. 3(1). Springer Netherlands, Aug. 1988, pp. 9-44.

[4] J. R. Millán, D. Posenato, and E. Dedieu, "Continuous-Action Q-Learning," in *Machine Learning*, vol. 49(2-3). Springer Netherlands, Nov. 2002, pp. 247-265.

[5] A. J. Smith, *Dynamic generalisation of continuous action spaces in reinforcement learning: A neutrally inspired approach*, Ph.D. dissertation, Division of Informatics, Edinburgh University, UK, 2001.

[6] S. Mahadevan and J. Connel, "Automatic programming of behavior-based robots using reinforcement learning," in *Artificial Intelligence*, vol. 55(2-3). Elsevier Science B.V., Jun. 1992, pp. 311-365.

[7] A.-H. Tan, N. Lu and D. Xiao, "Integrating Temporal Difference Methods and Self-Organizing Neural Networks for Reinforcement Learning with Delayed Evaluative Feedback," *IEEE Trans. on Neural Networks*, vol. 19(2), pp. 230-244, Feb. 2008.

[8] M. L. Littman, "Markov games as framework for multi-agent reinforcement learning", in *Proc. of the Eleventh Int. Conf. on Machine Learning*, Rutgers University, New Brunswick, NJ.. Morgan Kaufmann, Jul. 1994, pp. 157-163.

[9] L. Gambardella and M. Dorigo, "Ant-Q: A Reinforcement Learning approach to the traveling salesman problem," in *Proc. of the Twelfth Int. Conf. on Machine Learning*, Tahoe City, California. Morgan Kaufmann, Jul. 1995, pp. 252-260.

[10] C. Watkins and P. Dayan, "Q-Learning" in *Machine Learning*, vol. 8(3-4). Springer Netherlands, May 1992, pp. 279-292.

[11] C. F. Touzet, "Neural reinforcement learning for behavior synthesis" in *Robotics and Autonomous Systems*, vol. 22(3-4). Elsevier Science B.V., Dec. 1997, pp. 251-281.

[12] T. Kohonen, *Self organisation and associative memory*, 3rd ed. Springer-Verlag New York, Inc., 1989.
[13] D. R. Hofstadter, "Metamagical Themas: Computer Tournaments of the Prisoner's Dilemma Suggest How Cooperation Evolves" in *Scientific American*, vol. 248(5). Scientific American, Inc., May 1983, pp. 16-26.
[14] R. Axelrod, *The Evolution of Cooperation*. New York: Basic Books, 1984.

VLSI Design of Analog Neural Networks for Pattern Recognition

Dzmitry Maliuk

Abstract **– Analog implementation of neural networks offers advantages of small size, low power consumption and high speed. In this paper the design of analog synapses and neurons, the building blocks of any neural network, is described. These blocks are then tested on two network configurations, the 2-input perceptron and the discrete Hopfield network. Simulation results are used to verify the correct behavior of the analog circuits.**

Index Terms **– analog multiplier, analog neuron, perceptron, discrete Hopfield neural network.**

I. INTRODUCTION.

ANOLOG VLSI implementation of neural networks has received considerable attention from many researchers [1-3]. The major advantage is that analog implementation of basic computational components, such as multipliers and adders, in modern CMOS technologies is far more area efficient than digital implementation. This allows fitting many neurons on a single chip and achieving tremendous computational power. In contrast to software implementations, which sequentially emulate operation of neural networks, analog neural networks (ANN) utilize the inherent parallelism of neural models. Digital implementation of neural networks is also attractive for their parallelism, high precision and straightforward design, but they lack the compactness and power efficiency of ANN.

A simple McCulloch-Pitts neuron model is shown in Fig. 1. It has two types of building blocks: neurons and synapses. A synapse can be considered as a multiplier of an input signal value by the stored weight value. A neuron sums the output values of the connected synapses and compares this sum with a threshold value. Efficient implementation of these basic building blocks is essential for large single-chip networks. The most compact implementation of a synapse reported in the literature uses a single floating gate transistor [4]. This approach holds a great potential, especially for small ultra-low power networks. However, the technology is not now mature enough for practical implementations in standard CMOS processes, which would require high voltages and a long time for accurate floating gate programming.

We present architecture of a McCulloch-Pitts neuron, which can be implemented in standard CMOS technology. The synapse is an analog multiplier, while the neuron consists of a current to voltage converter and a high gain amplifier. In addition, two example networks, a 2-input perceptron and a discrete Hopfield's network, composed of the designed building blocks, are demonstrated. The schematic of the design has been implemented and simulated in Cadence Analog Design Environment. The design is targeted for TSMC 0.18ŭ technology, the simulation was carried out with the corresponding transistor models.

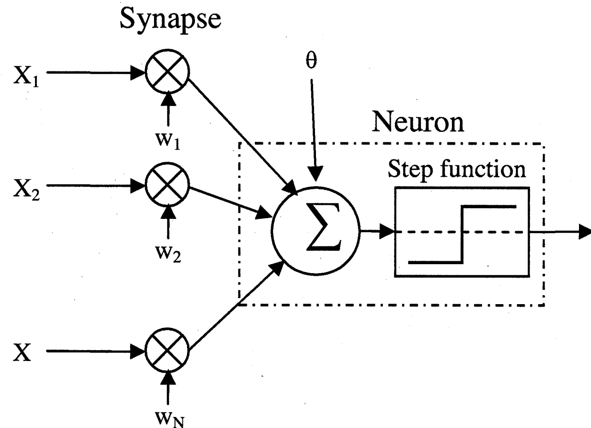


Figure 1. McCulloch-Pitts neuron model

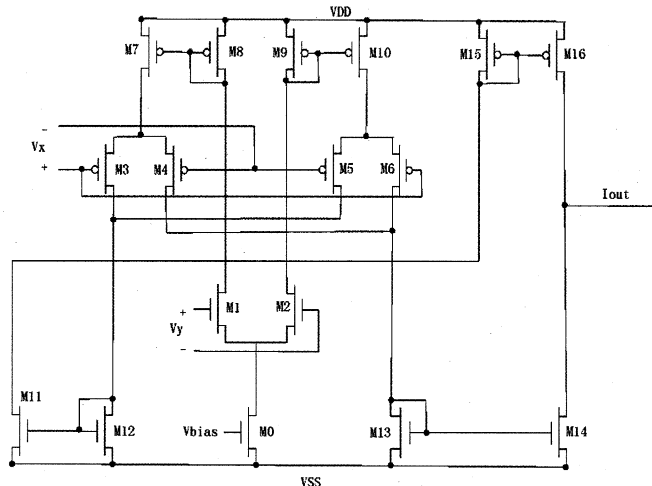


Figure 2. Four-quadrant CMOS analog multiplier

II. ANN CIRCUITS.

A. *The Synapse Circuit.*

A four-quadrant CMOS analog multiplier [5] has been chosen to perform the synapse operation. The schematic diagram of this circuit is shown in Fig. 2. The inputs to the circuit are two differential mode voltage signals V_x and V_y, and the output is the current signal I_{out} proportional to the product of the inputs. The basic element is a Gilbert six-transistor cell composed of three differential pairs (transistors M1-M6). The differential current produced by this cell is

nonlinear except for a small range of input voltages around the origin, where it can be approximated by

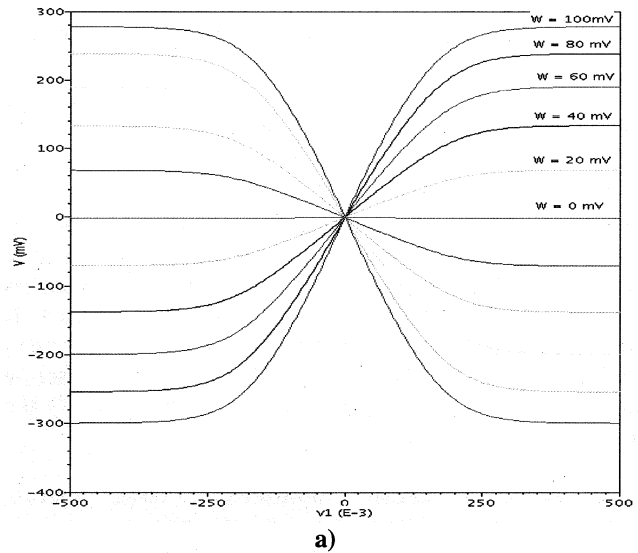
$$I_{OUT} \cong \sqrt{2k_n k_p}\, V_x V_y$$

Here, k_n and k_p are the transconductance coefficients of nmos and pmos transistors respectively, V_x and V_y – differential input voltages. The linearity requirement sets the constraint on the maximum input voltage to be $|V| \ll \sqrt{I_{ss}/k}$, where I_{ss} is the biasing current set by M0. Therefore, the dynamic range of the input voltages can be increased either by having relatively large current I_{ss} or choosing a small k which means that the input transistors are narrow and long.

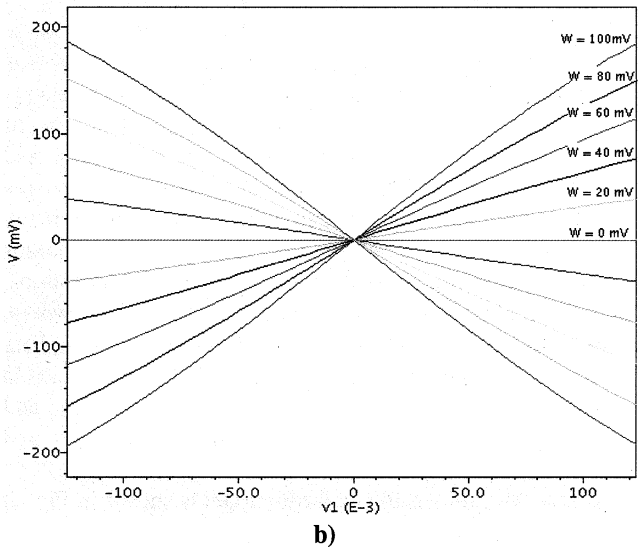
The parameters of the circuit have been calculated based on the specifications listed in Table 1. The input-output characteristics for a number of weight values (W) are shown in Fig. 3. For the purpose of simulation the output current is converted into voltage. The slopes of the traces are proportional to the weight values. In part a) of Fig. 3 the differential input v1 sweeps across a wide voltage range. For large V, the circuit enters the saturation region and cannot be used for multiplication. As a result, only a small range of input voltages around the origin, where the characteristic is close to linear, are of relevance. Such a small range from -100mV to 100mV is shown in part b) of Fig. 3. The same restrictions apply to the differential weight input, the useful range of which has been chosen to be the same as the range of V.

Table 1. Multiplier Specification

Parameters	Range
Input Range	+/- 100 mV
Output Range	+/- 160 mV
V_{dd} /V_{ss}	0.9 V/ -0.9 V



a)



b)

Figure 3. Characteristics of CMOS multiplier: a) large signal; b) small signal.

B. The Neuron Circuit.

The function of a neuron is to add together the outputs of its synapses and compare it with its threshold. Since the outputs of individual neurons are currents, the summation operation is implemented as current summation. The threshold current is generated by a simple voltage to current converter, shown in Fig. 4a, and added together with synapse currents. The resultant current is then converted to a voltage by a high gain amplifier (Fig 4b), which in effect outputs high voltage for the positive input current and low voltage for the negative. It is implemented by cascading a linear current to voltage converter with two inverters.

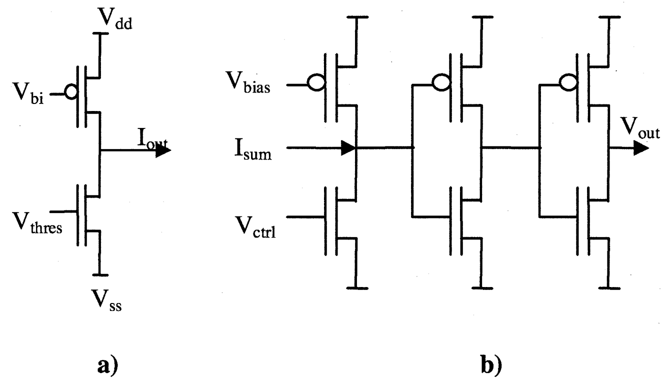


Figure 4. Neuron circuits: a) V->I converter; b) I->V threshold converter.

III. TWO-INPUT ANALOG PERCEPTRON.

To test the designed analog blocks we have assembled a simple neural network to perform a logical AND operation. This ANN consists of one neuron and two synapses (Fig 5). Such configuration is very easy to train, since we can directly extract the values for the weight and threshold coefficients from a separating line between two classes. The inputs must first be normalized to fit into the allowable multiplier input values range. Fig. 6 shows four normalized logical inputs as well as some other inputs closer to the separating line to test accuracy of the threshold. The sequence of input vectors as they are applied to the network is shown by the arrows. The simulation results are illustrated in Fig 7.

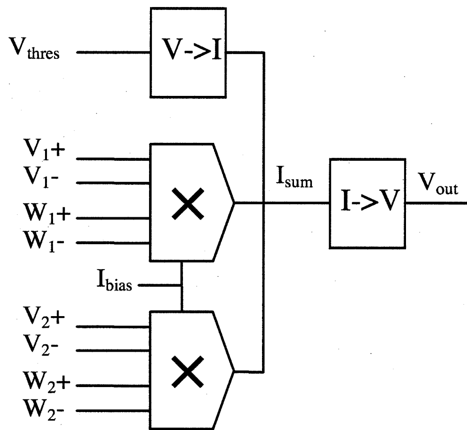


Figure 5. Block diagram of the two-input perceptron.

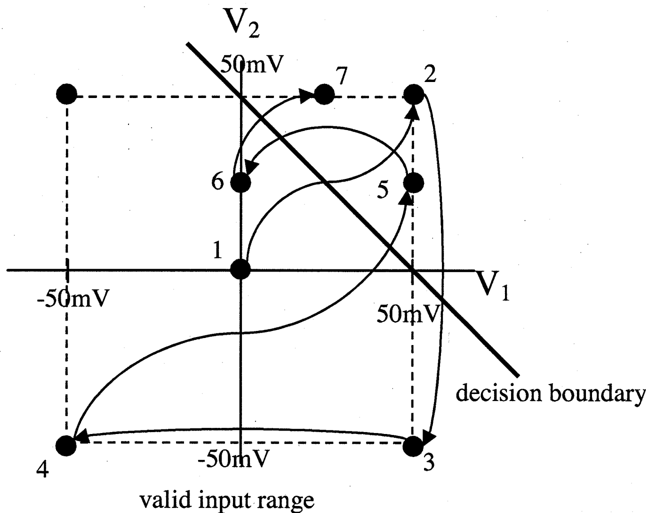


Figure 6. Sequence of input test vectors.

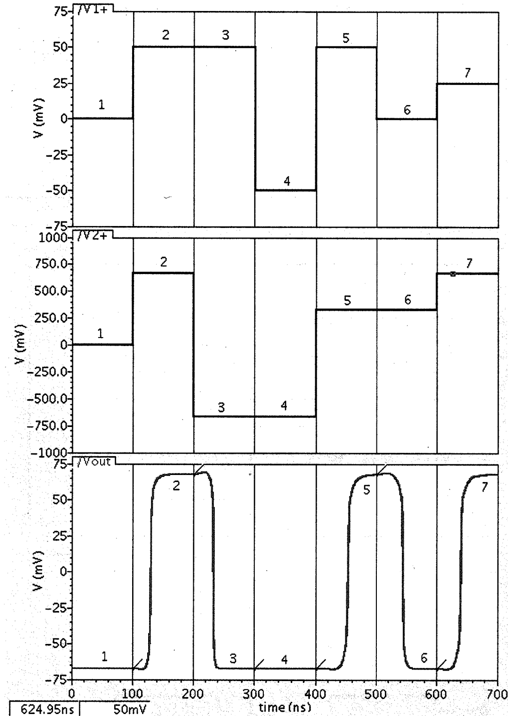


Figure 7. Simulation results of the two-input perceptron.

IV. DISCRETE HOPFIELD NEURAL NETWOK.

A more complicated network built from the designed analog blocks is shown in Fig. 8. This is a version of an analog Hopfield network with McCulloch-Pitts neurons having two distinct states. This circuit belongs to the class of mixed-signal circuits, incorporating digital and analog devices on the same chip. The network is fully interconnected, i.e. each neuron has connections with all other neurons. The state of the circuit is characterized by a state vector, which is the output of the four registers. The dynamics of the circuit is controlled by the clock signal, which updates the state vector and initiates further neural processing. The rate of the clock is limited by the longest path of the analog signal and depends on implementation details.

The required operation of this network is achieved by programming the weight coefficients of individual neurons, which depend on a set of fundamental memories we desire to store. The weight matrix of this network can be calculated by the following formula [6]

$$\boldsymbol{W} = \frac{1}{N}\left(\sum_{m=1}^{M} \xi_m \xi_m^T - MI\right)$$

where ξ_m are fundamental memories, N = 4, M is the number of fundamental memories. For the purpose of demonstration the set of fundamental memories was chosen to be $\xi_1 = [1\ \ 1\ \ 1-1]^T$, $\xi_2 = [-1\ \ 1-1\ \ 1]^T$. The corresponding weight matrix after normalization to conform to the input voltage range requirements is

$$W = \begin{bmatrix} 0 & 0 & 25 & -25 \\ 0 & 0 & 0 & 0 \\ 25 & 0 & 0 & -25 \\ -25 & 0 & -25 & 0 \end{bmatrix} mV$$

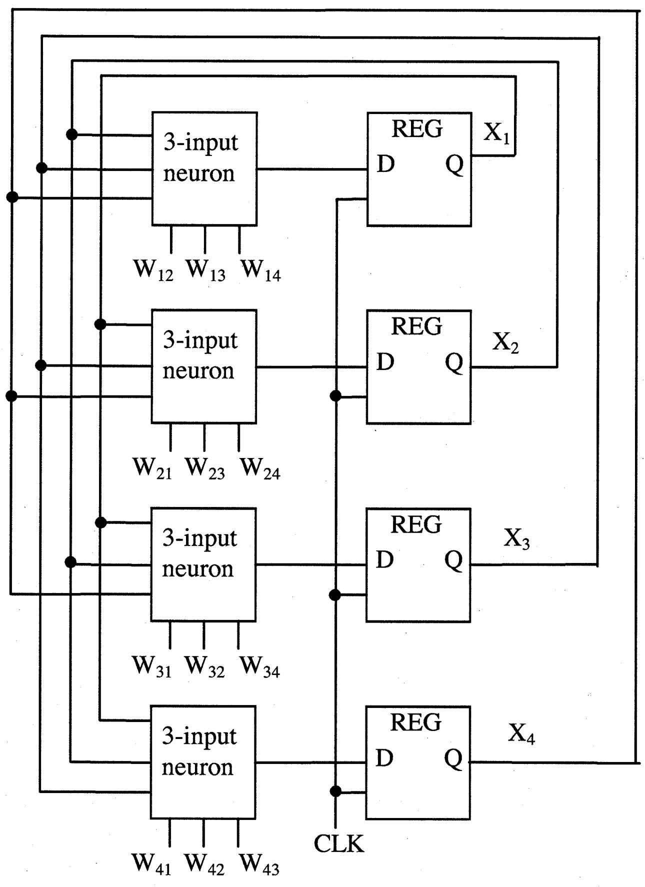
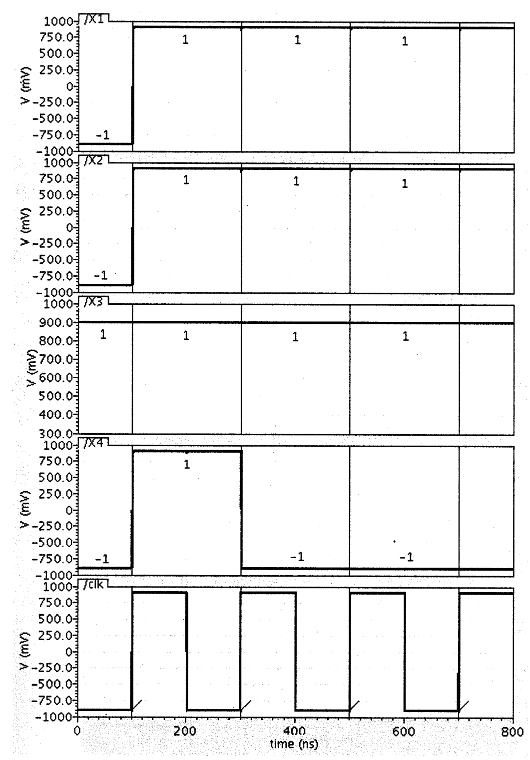
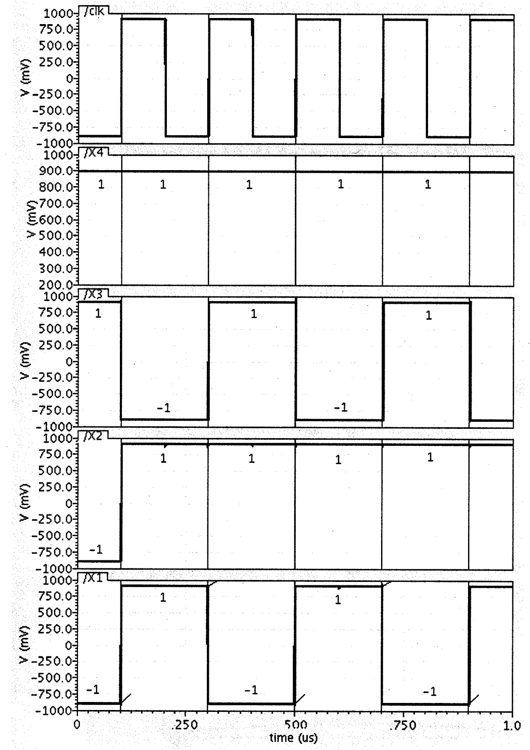


Figure 8. Discrete analog Hopfield network.

Fig. 9 demonstrates the results of simulation for the given weight matrix for two initial states: $\boldsymbol{X}_0 = [-1 - 1 \ \ 1 - 1]^T$ and $\boldsymbol{X}_0 = [-1 - \ 1 \ \ 1 \ \ 1]^T$. The nodes from X1 to X4 represent the state vector. It is used to evaluate the outputs of the neurons, which will update the state vector on the next rising edge of the clock. In Fig. 9 the state of each register during a clock cycle is denoted by the corresponding number. It can be noted that in the first case the network converges to one of its fundamental memories, namely $\boldsymbol{\xi}_1 = [1 \ \ 1 \ \ 1 - 1]^T$. In the second case the output keeps oscillating between two states and never converges to any of the fundamental memories. This network has also been simulated in MATLAB to verify the output sequence of the analog version.



a)



b)

Figure 9. Simulation results of the Hopfield neural network for two initial conditions: a) X_0=[-1 -1 1 -1]^T; b) X_0=[-1 -1 1 1]^T

V. Conclusion.

We presented the design of analog synapses and neurons, as well as two example neural networks built from these components. The simulation results demonstrate feasibility of the designed analog circuits as building blocks of larger neural networks. In particular, the network configurations presented can be extended to higher dimensions. However, it would require more careful design to retain signal integrity, because high fan-in or fan-out analog signals are more susceptible to noise and distortion.

References

1. M. Verleysen, P. Jespers, "An Analog VLSI Implementation of Hopfield's Neural Network", IEEE MICRO, vol. 9, no. 6, pp. 46-55, Dec 1989.
2. C.Y. Wu, J.F. Lan, "CMOS Current-Mode Neural Associative Memory Design with On-Chip Learning", IEEE Trans. On Neural Networks, vol.7, no.1, Jan. 1996.
3. J. Ghosh, P. Lacour, S. Jackson, "OTA-Based Neural Network Architectures with On-Chip Tuning of Synapses", IEEE Trans. On Circ. and Syst. II, val. 41 no.1, pp. 49-58, Jan 1994.
4. P. Hasler, C. Diorio, B. Minch, C. Mead, "Single Transistor Learning Synapses", IEEE Trans. On Electron Devices, vol. 43, no. 11, pp. 1972-1980, Nov. 1996.
5. J. N. Babanezhad, G. C. Temes, "A 20-V Four-Quadrant CMOS Analog Multiplier", IEEE Journal of Solid-State Circuits, vol. sc-20, no. 6, pp. 1158-1168, Dec. 1985.
6. S. Haykin, "Neural Networks: A Comprehensive Foundation", 2nd edition, Prentice Hall, 1999.

Using Neural Networks to Estimate Volatility of Financial Returns

Wedzerai V Munyengwa

***Abstract* - Predicting the volatility of returns from financial instruments is a very difficult, but nonetheless rewarding, endeavor. Various linear methods have been used to predict volatility using past values of some variable. In this paper I develop a Neural Network Model that predicts both returns and volatility of the monthly returns on S&P500 stock index. We assume that the system that generates volatility has some memory of the recent past and responds to news which is necessarily random. By analyzing the serial correlations of volatility time series we confirm the memory assumption, i.e. future volatility depends on the its own historical values. By analyzing the residuals (actual less estimated value) of stock returns estimated by our neural network, we also confirm that the error is i.i.d (normally distributed) noise. The Neural Network model performs nearly the same as the GARCH model, but is a lot simpler and easy to use.**

***Index Terms*—Volatility, Neural Network**

I. INTRODUCTION Most models used to price financial instruments invariably assume a constant volatility of returns (log returns to be specific). However studies have shown that volatility itself varies with time, and depends on random noise (reaction to news) its own past values [1]. GARCH models have been developed to incorporate these findings [1]. Studies by behavioral economists have also shown that financial markets exhibit excess volatility, i.e. realized volatility significantly exceeds that which can be explained by efficient market hypothesis [2]. They offer explanations based on irrational behavior by investors, and on feedback models [3]. This provides motivation to use Neural Networks to estimate volatility, and financial returns.

We will use S&P500 stock index data to train my neural network model, using the back propagation algorithm. We will also develop a GARCH model to model the volatility time series and compare its performance to that of the neural network. Fig 1 is a plot of the stock price time series of the S&P500 Composite index. Let $p(t)$ be the stock price in month t. Then we calculate log return for month t as $r(t) = logP(t) - \log P(t-1)$, and the volatility as $\sigma(t) = var(r(t), \ldots r(t-12))$. Figure 2 shows a plot of autocorrelation of the volatility time series versus lag. There is significant autocorrelation for high lag values implying that the system generating this time series has significant memory.

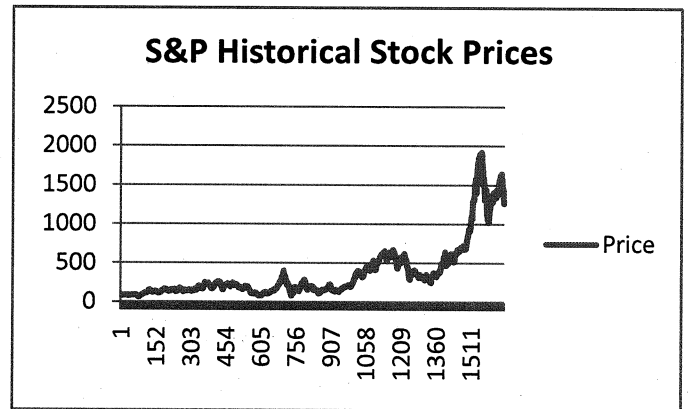


Figure 1: Historical values of the S&P500 equity index

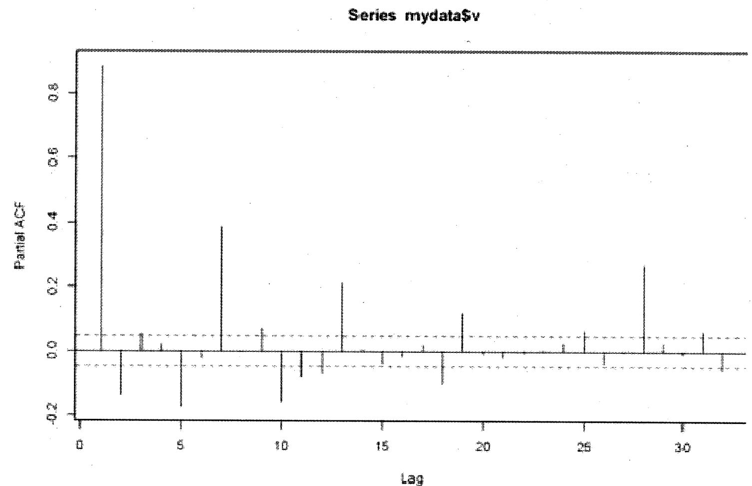


Figure 2: Partial Autocorrelations

GARCH (p, q) Model

A Generalized Autoregressive Conditional Heteroskedasticity (GARCH [p, q]) model is represented in equation 1, where $\sigma(t)^2$ is the conditional variance, $u(t) = r(t) - E[r(t)]$ is the residual of the log returns, and $E[\,]$ is the expectation operator.

$$\sigma(t)^2 = \kappa + [\textstyle\sum_{i=1}^{p} \alpha_i u(t-i)] + [\sum_{j=1}^{q} \beta_i \sigma(t-j)^2] \quad (1)$$

$$\kappa > 0,\ \alpha_i \geq 0,\ \beta_i \geq 0$$

p is the number of lags of residuals used, and q is the number of lags of volatility. To keep things simple we assume that the volatility follows a GARCH (3, 3) process, i.e. p=3, and q=3.

$$\sigma(t)^2 = \kappa + \alpha_1 u(t-1)^2 + \alpha_2 u(t-2)^2 + \alpha_3 u(t-3)^2 + \beta_1 \sigma(t-1)^2 + \beta_2 \sigma(t-2)^2 + \beta_3 . \sigma(t-3)^2 \quad (2)$$

Equation (2) models the volatility$\sigma(t)^2$. $u(t)$ is the residual (which is the error in the estimation of $r(t)$ by conditional expectation $E[r(t)]$.

We assume that $u(t)$ follows normal distribution. Then $E[r(t)] = 0$, and $var[r(t)] = \sigma(t)^2$. Thus

$$f(u_t|u_{t-1},\dots,u_1) = \frac{1}{\sqrt{2\pi\sigma(t)^2}} e^{-\frac{u(t)^2}{2\sigma(t)^2}} \quad (3)$$

$$f(u_n,\dots,u_1) = f(u_{n-1},\dots,u_1)f(u_n|u_{n-1},\dots,u_1) \quad (4)$$

Iterating formula (4) gives

$$f(u_n,\dots,u_1) = f(u_1)f(u_2|u_1)f(u_3|u_2,u_1)\dots f(u_n|u_{n-1},\dots,u_1) \quad (5)$$

Taking natural logs of (5) gives the log likelihood function

$$L(\omega,\alpha_1,\alpha_2,\alpha_3,\beta_1,\beta_2,\beta_3) = -\left(\frac{n}{2}\right)\log(2\pi) - \left(\frac{1}{2}\right)\sum_{t=1}^{n}\{\log(\sigma_{t-1|t-2}^2))\} + u_t^2/\sigma_{t|t-2}^2 \quad (6)$$

We then estimate the parameters $\omega,\alpha_1,\alpha_2,\alpha_3,\beta_1,\beta_2,\beta_3$ that maximize the log likelihood by using optimization methods like Lagrange method.

Neural Network Model

I use a neural network model to estimate both returns and conditional variance. I design a multilayer perceptron with three inputs. Let $x(t) = [r(t-1), u(t-1)^2, \sigma(t-1)^2]^{tr}$ be the input vector, and $y(t) = [r(t), \sigma(t)^2]$ be the output vector. The neural network is shown in fig 3. The activation/transfer function is tanh for all neurons except for the one that outputs r(t) (with linear transfer function) [4].

The MLP algorithm should be such that the log likelihood of the data is maximized [5]. After taking the natural logarithms of both sides in equation (3), we get

$$\log[P(u(t)|u(t-1)..)] = -\left(\frac{1}{2}\right)\log(2\pi) - \left(\frac{1}{2}\right)\log(\sigma(t)^2) - u(t)^2/2\sigma(t)^2 \quad (7)$$

We can ignore the constant $\left(\frac{1}{2}\right)\log(2\pi)$ and view the problem as one of minimizing the cost function C, instead of the usual error function $e(n)$.

$$C = \sum_i\{\left(\frac{1}{2}\right)\log(\sigma(t)^2) + \frac{u(t)^2}{2\sigma(t)^2}\} \quad (8)$$

r(t − 1)
u(t − 1)²
σ(t − 1)²
r(t)
σ(t)²

Figure 3

Back-propagation weight updating algorithm

Assuming a linear transfer function for the neuron that outputs $r(t)$ and sigmoid transfer functions for all other neurons, weights are changed according to the following delta formula.

$$\Delta w_{ji}(t) = -\eta\frac{\partial C}{\partial w_{ji}(t)}$$

where C is given in equation 8.

Let $y_i(t)$ be the output from neuron i, $w_{ji}(t)$ the weight of the connection from neuron i to neuron j, $v_j(t)$ the local signal of neuron j, b_j the bias to neuron j, φ_j is the transfer function for neuron j, and $d(t)$ the desired/realized return at time t.

Updating weights for connections to neuron that output r (t):

$$u(t) = d(t) - r(t)$$

$$r(t) = \varphi_j(\left[\sum_i w_{ji}(t)y_i(t)\right] + b_j)$$

$$\frac{\partial C}{\partial w_{ji}(t)} = \frac{1}{2\sigma(t)^2}\frac{\partial u(t)^2}{\partial w_{ji}(t)}$$

$$\frac{\partial u(t)^2}{\partial w_{ji}(t)} = -2u(t)\frac{\partial r(t)}{\partial w_{ji}(t)} = -2u(t)y_i(t)$$

$$because\ \varphi_j\ is\ linear$$

Therefore

$$\Delta w_{ri}(t) = \eta\left(\frac{1}{\sigma(t)^2}\right)[d_i - r(t)]y_i(t)$$

Updating weights for connections to neuron that output$\sigma(t)^2$:

$$\Delta w_{ji}(t) = -\eta\left(\frac{1}{2\sigma(t)^2}\right)\{[d_i - r(t)]^2 - \sigma(t)^2\}h_{ji}(t)$$

$$\sigma(t)^2 = \varphi_j(\left[\sum_i w_{ji}(t)y_i(t)\right] + b_j)$$

$$\frac{\partial C}{\partial w_{ji}(t)} = \left(\frac{1}{2}\right)\left[\left(\frac{1}{\varphi_j\left(v_j(t)\right)}\right) - [d(t) - r(t)]^2/\varphi_j^2\left(v_j(t)\right)\right]\frac{\partial\varphi_j\left(v_j(t)\right)}{\partial w_{ji}(t)}$$

$$\frac{\partial C}{\partial w_{ji}(t)} = \left(\frac{1}{2\varphi_j^2\left(v_j(t)\right)}\right)\left[\varphi_j\left(v_j(t)\right) - [d(t) - r(t)]^2\right]\frac{\partial\varphi_j\left(v_j(t)\right)}{\partial w_{ji}(t)}$$

$$\varphi_j\left(v_j(t)\right) = tanh([\sum_i w_{ji}(t)y_i(t)] + b_j)$$

$$therefore\ \frac{\partial\varphi_j\left(v_j(t)\right)}{\partial w_{ji}(t)} = y_i(t)(1 - tanh^2(v_j(t))$$

$$\frac{\partial \varphi_j\left(v_j(t)\right)}{\partial w_{ji}(t)} = y_i(t)(1-\varphi_j{}^2(v_j(t)))$$

$$\Delta w_{ji}(t) = -\eta\left(\frac{y_i(t)}{2\sigma(t)^4}\right)[\sigma(t)^2-[d(t)-r(t)]^2][1-\sigma(t)^4]$$

Updating weights for connections to the rest of the neurons:
We use the usual back propagation method of calculating weights for connections to non-output neurons.

RESULTS

Below are plots of estimates of conditional variances using MLP then GARCH.

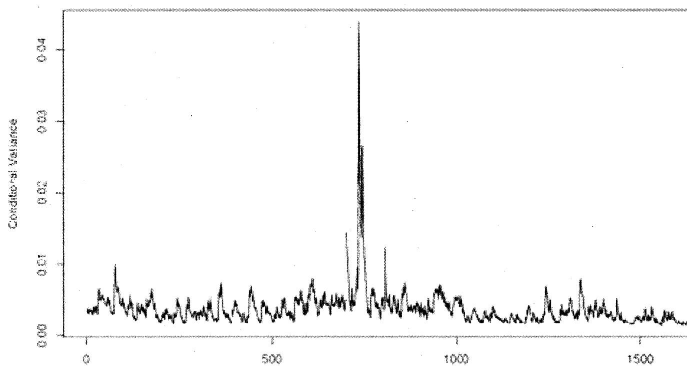


Figure 4: Estimates of Conditional Volatility using MLP

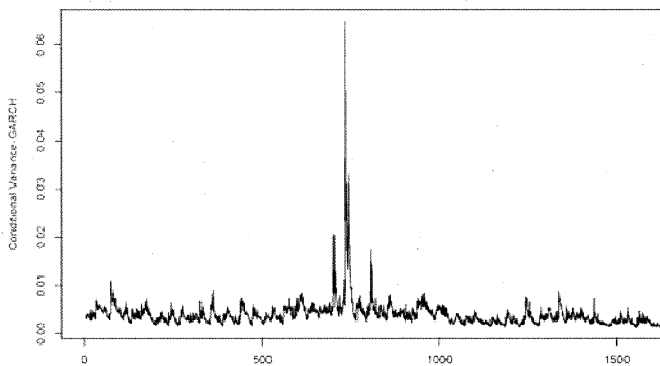


Figure 5: Estimates of Conditional Variance using GARCH

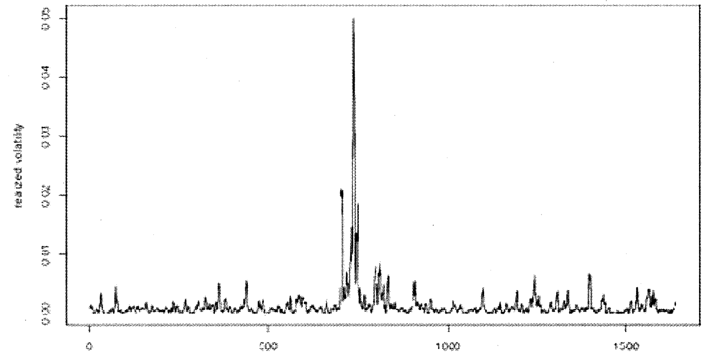


Figure 6: Actual Variance

As can be seen from the graphs, both methods did fit the data satisfactorily. Neural Networks achieved that using many parameters (weights in the MLP) but the GARCH model uses only 7 parameters. This is not a big issue since the data set I use is very large.

Figures 7 and 8 below plot the estimates of the residuals u(t).

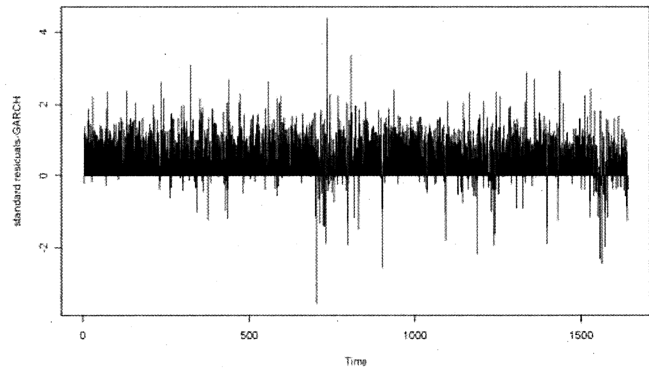


Figure 7: Standard Residuals using GARCH

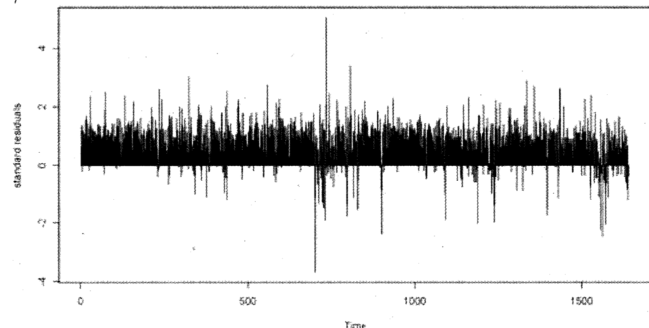


Figure 8: Standard Residuals using MLP

The Normal Q-Q plot of the residuals from the MLP estimation in figure 9 below shows that the residuals estimates follow normal distribution. This confirms what we assumed in our model.

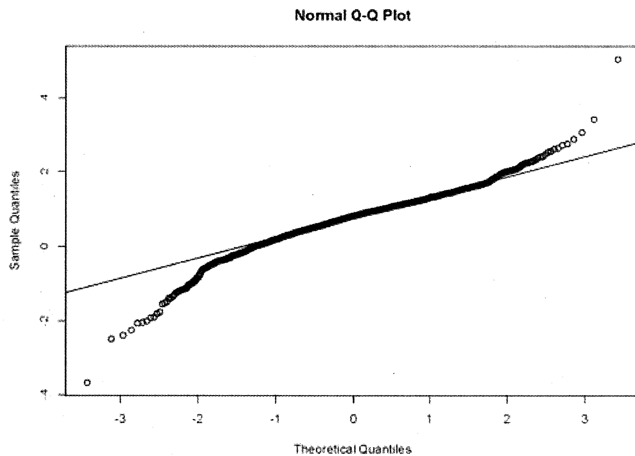


Figure 9: Normal Q-Q Plot for residuals: MLP

Conclusion

Even though the MLP was outperformed by GARCH, it gave satisfying results. The neural network was able to estimate the volatility to almost the same extend as the GARCH model. The Neural network is also not difficult to implement and we can train it with large amounts of historical data.

REFERENCES

M. Casdagli, S. Eubank, J.D. Farmer, and J. Gibson, "State Space Reconstruction in the Presence of Noise.'' *Physica* D, vol. 51D, pp.

W.L. Buntine and AS. Weigend, "Bayesian Backpropagation." ***Complex Systems,*** vol. ***5,*** 52-98, 1991 Clerk Maxwell, *A Treatise on Electricity and Magnetism 7,* 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp. 68-73.

REFERENCES

[1] Robert F. Engle. "Autoregressive Conditional Heteroscedasticity with Estimates of Variance of United Kingdom Inflation", *Econometrica* 50:987-1008, 1982. *(the paper which sparked the general interest in ARCH models)*

[2] **Shiller, R. J.** (1990) Market Volatility - pages 1-4, 71-76, 197- 214, MIT Press, London
Schiller,R.J. (1981) Do Stock Prices Move Too Much To Be Justified by Subsequent Changes in Dividends? *The American Economic Review*, Volume 71, June, No.3 pages 421-435

[3] Shiller, R.J., 2005a. *Irrational Exuberance*, 2nd ed. (Princeton University Press, Princeton, NJ).

[4] A. Weigend, and D. Nix "Error bars for non-linear regression," *Advances in Neural Information Processing 7 (NIPS 94),* pp489-496, 1994.
M. Casdagli, S. Eubank, J.D. Farmer, and J. Gibson, "State Space Reconstruction in the presence of Noise.'' *Physica* D, vol. 51D, pp

[5] W.L. Buntine and AS. Weigend, "Bayesian Backpropagation." ***Complex*** **Systems,** vol. **5,** 52-98, 1991 Clerk Maxwell, A Treatise on Electricity *and Magnetism 7,* 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp. 68-73

Neural Networks Playing Tic Tac Toe Trained by Backpropagation

Zachary Murez

Abstract—**The use of feed forward neural networks (nets) to play the simple game of tic tac toe was studied. There have been many previous experiments using genetic algorithms to train nets, but for this study, backpropagation with sample moves was employed. The nets were tested against random players and were compared to players that use look up tables of the sample moves the nets had been trained with. Playing ability, as well as playing speed were analyzed.**

Index Terms—**Neural Networks, Tic Tac Toe, Artificial Intelligence.**

I. GAME

Tic Tac Toe is a simple game comprised of a nine element playing board. The nine squares are arranged in three rows of three columns. It is normally played between two individuals, where player one is known as "X" and player two as "O" on the board. The rules are fairly limited; each player gets one move at a time. With each move, the player fills one of the nine squares with his (or her) respective marker ("X" or "O"), which signifies that he (or she) has captured the square. The objective is to capture three squares in a row, column or diagonal.

II. HISTORY

There is a simple set of rules that allow optimal play, whereby player 2 can always force a draw. However, having a machine learn to play well requires a more complicated algorithm. The first successful attempt at this was Donald Michie's Machine Educable Noughts And Crosses Engine (MENACE), devised in 1960. This used 300 matchboxes, each corresponding to a unique board state. Each matchbox contained beads of 9 colors, representing the possible moves. The percentage of beads in a matchbox represented the probability that that move was a good one. A move was made by selecting a bead at random from the box that corresponded to the current board state. MENACE started out with equal numbers of each color bead. It played a game, and if it won, the beads were replaced; if it lost, the beads were discarded. Thus, over time, moves that led to losses became less likely and moves that led to wins became more likely. MENACE eventually converged to optimal play. Since MENACE, there have been many successful implementations using neural nets. Most are trained using a genetic algorithm. In this case, a large set of neural nets was created with the weights initialized to random values. The fitness of each was measured by games won and lost. The fittest survived, while the rest were eliminated. Then the fittest reproduced, during which recombination and mutation occurred. The process was repeated until convergence, or a limit on iterations, was reached.

III. MOTIVATION

When a human is learning to play a game, he learns by two processes. The first one is by playing and gaining experience as to what leads to wins and losses, as modeled by the genetic algorithm. The second way is by being shown examples of good moves by a teacher. The latter could be modeled by using backpropagation. A human who learns by only one of the two methods limits his capability. The same likely holds true for neural nets, and thus the most powerful algorithm would be one that uses both methods. Since it has already been demonstrated that neural nets can be trained to play tic tac toe by genetic algorithms, the goal was to show that they could also be trained by backpropagation. Future work would combine the two methods by alternating between modes of learning.

IV. STRUCTURE OF THE NET

Each square of the board was mapped to one of 9 inputs. Each input was a (-1) denoting "O", 1 denoting "X", or 0 denoting blank. A feed forward net with one hidden layer containing ten neurons was fully connected to the inputs and the 9 output neurons. Each output was then mapped back to a square of the board. The use of this structure was an arbitrary decision that was experimented with later. The activation functions were sigmoid functions (1/[1+e^x]). Thus, each output was a number between 0 and 1. The move specified by the net corresponded to the neuron with the largest output that was unoccupied. For example, if output neuron one had a value .8 and neuron two had a value .5, the move was square one, unless square one was already occupied. The net's weights were initialized to 0. The net was then trained. It was presented with a board configuration, and it calculated the outputs. It was next presented with the desired outputs, which were 0 for all neurons, except the one that corresponded to the optimal move, which had the value 1. Backpropagation was finally used to update the weights. Training continued for between 1000 and 20,000 iterations.

V. TEST 1: GENERALIZATION ON EXAMPLES SHOWN TO PLAYER 2

1) Random vs. Random
As a benchmark, both players were tested making random moves. This was achieved by selecting a random square from those that were unoccupied.
2) Random vs. Partially Optimal
As another benchmark, player 1 played randomly and player 2 played partially optimally. This was achieved by searching through a subset of board configurations with known optimal moves. Whenever player 2 saw a board configuration which was in its lookup table, it made the optimal move; otherwise it made a random move.
3) Random vs. Neural net
The net was trained using the same subset of known examples as the previous test.

VI. RESULTS 1

Players were tested on 1000 games. Of the 2000 possible board configurations for which it was player 2's turn to act, 1000 of them were in the lookup table. The neural net was trained in increments of 500 examples, and then tested until the desired 20,000 iterations were attained.

Results are displayed in Figures A and B below. Figure A shows that random player 1 won the most against random player 2. As training of the neural net increased, player 1 won less and approached the amount won against the partially optimal player. However, player 1 won the least when playing against the partially optimal player. Figure B shows that player 2 won the least when playing random. The net won more as it was trained more, but the partially optimal player won the most. This indicates that although convergence occurred, there was no generalization to cases that it had not been shown. If it had, then after a certain number of iterations, player 1 would have won less and player 2 would have won more than the partially optimal player.

VII. TEST 2: STRUCTURE OF THE NET

Lack of generalization indicated that there were too many neurons, or not enough hidden layers. Thus experimentation was done using different size nets. The input layer and output layer remain the same. Player 1 was always a random player and player 2 was varied. In each test the net was trained for 10000 iterations and then played 1000 games.

VIII. RESULTS 2

TABLE 1 HERE

Nets with two hidden layers performed slightly better than those with one such layer. However, due to the large variance in scores (because of the random player), it was hard to tell which structure was actually best. Interestingly, the performance did not decay with fewer neurons until the hidden layer contained 2 neurons or less. However, significant generalization still did not occur.

IX. TEST 3: GENERALIZATION OF PLAY OF PLAYER 1 FROM TRAINING AS PLAYER 2

Since the net did not generalize as player 2, it was tested to see if it could be trained using examples where it is player 2's turn to act, and then played as player 1.
1) Random vs. Random
2) Neural Net vs. Random

X. RESULTS 3

Players were tested on 1000 games.
The subset of known board configurations was 1000 of approximately 2000 for which it was player 2's turn to act. The neural net was trained with 500 examples in each iteration. Its structure was the original one (1 hidden layer with 10 hidden neurons). Results are displayed in Figure C below. The neural net definitely became a better player 1 as it was trained as player 2. This shows that it had generalized.

XI. PERFORMANCE

Since the net did not play better than the semi-optimal player, they were both tested for performance characteristics. The question was whether a neural net could decide on a move faster than searching through a table of known moves. This would be a big advantage to a net, since they both play about the same.

XII. TEST 4: SPEED

10000 games were played and the duration of time spent playing was recorded. The time required to train the neural net was not included, since only playing speed was being considered. The code was written in Microsoft Visual FoxPro Version 9 (VFP). This is a programming language known for its speed in database lookups. One of the key features of VFP is its use of Rushmore Optimization, which uses indexes to speed up lookup time dramatically. On the other hand, it is not very powerful at doing math computations as is required for implementation of the neural net. Nevertheless, some interesting results were obtained.

XIII. RESULTS 4

TABLE 2 HERE

It was found that, as expected, random moves were by far the fastest. Interestingly, however, the neural net played much faster than searching through the database, even with Rushmore Optimization.

XIV. CONCLUSIONS

The neural net generalized to situations as player 1 that were similar to those shown to it as player 2. However, it was not able to generalize to situations it was not shown as player 2. Thus, optimal strategy was not achieved. On the other hand, had it been trained using all 2000 possible situations for player 2, it would have been able to play optimally as both player 1 and player 2. Nevertheless, it was found that once a neural net was trained, it could play much faster than searching through the examples.

XV. FURTHER WORK

More experimentation could be done with the structure of the neural net to improve generalization; however this is unlikely to lead to any new insight. A major problem with this design is that the input is only a 9D vector and does not contain any topological information about the board being a grid. Designing the neural net in a more mathematically efficient language, such as Matlab, would most likely increase playing speed. Combining this type of learning with genetic learning would lead to more powerful learning. The end goal would be to apply these strategies to a more complicated game, such as checkers.

REFERENCES

[1] Wikipedia. "tic tac toe"

[2] Wikipedia. "Donald_Michie"

[3] Noughts and Crosses (Tic Tac Toe) using Artificial Neural Networks. http://www.adit.co.uk/html/neural_networks.html

[4] S. Hayken, Neural Networks: A Comprehensive Foundation.

[5] Training an artificial neural network to play tic-tac-toe. Siegel. http://homepages.cae.wisc.edu/~ece539/project/f01/siegel.pdf

Table 1: Structure Analysis

Player 2	Player 2 score
Random	285
Partially Optimal	610
1 hidden layer (10 neurons)	563
1 hidden layer (7 neurons)	562
1 hidden layer (4 neurons)	568
1 hidden layer (3 neurons)	572
1 hidden layer (2 neurons)	564
1 hidden layer (1 neurons)	380
2 hidden layers (5 neurons) (6 neurons)	543
2 hidden layers (4 neurons) (5 neurons)	591
2 hidden layers (5 neurons) (4 neurons)	556
2 hidden layers (4 neurons) (4 neurons)	587
2 hidden layers (3 neurons) (3 neurons)	530
2 hidden layers (3 neurons) (6 neurons)	596
2 hidden layers (6 neurons) (3 neurons)	563
2 hidden layers (2 neurons) (6 neurons)	587
2 hidden layers (6 neurons) (2 neurons)	593

Table 2: Performance

Players	Time (seconds)	Score
Random vs. Random	2.36	5791-2906
Random vs. Optimal without Rushmore Optimization	39.94	2804-5934
Random vs. Optimal with Rushmore Optimization	24.66	2763-5992
Random vs. Neural Net with 1 hidden layer (4 hidden neurons)	14.16	3123-5964

Figure A:

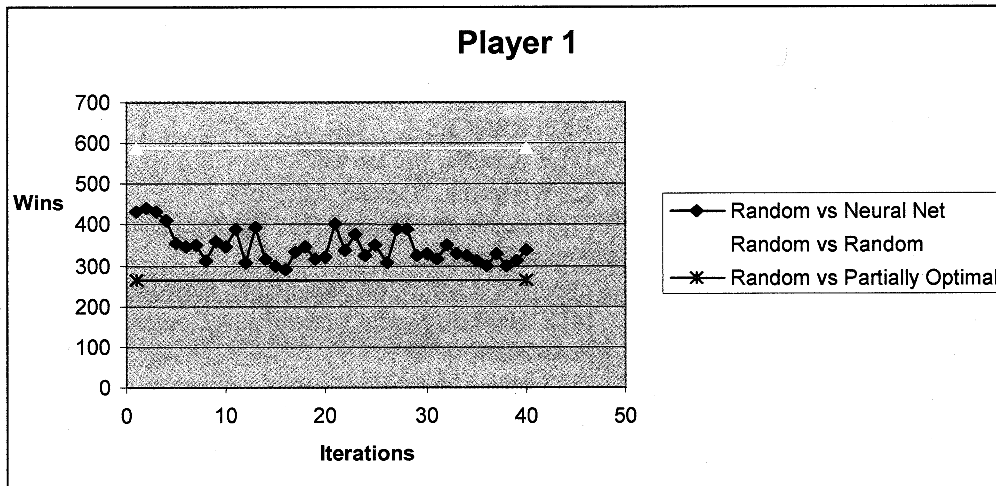


Figure B:

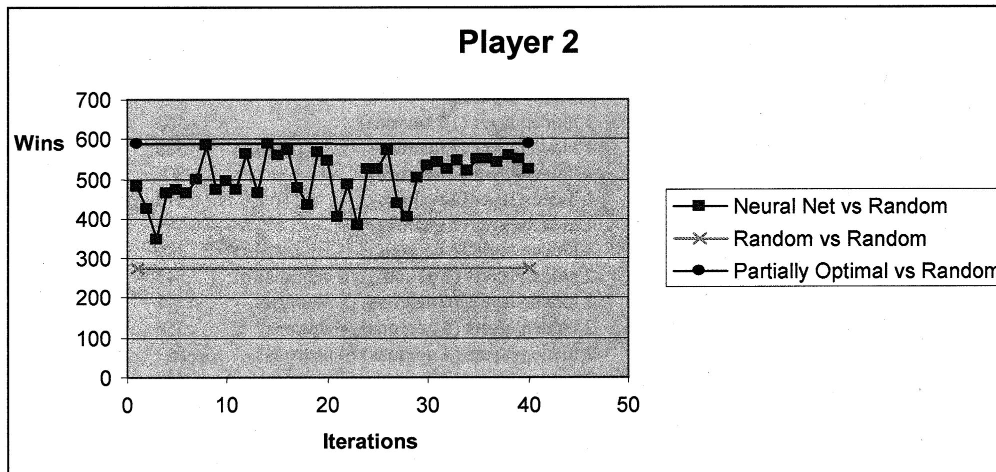
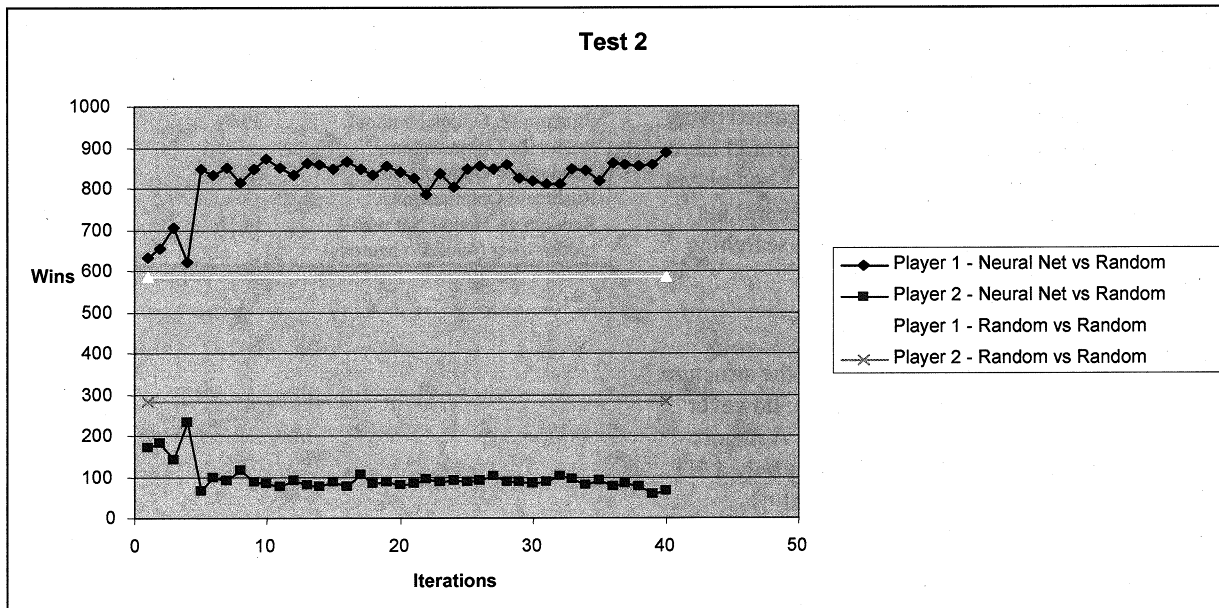


Figure C:



Hopfield Network for Clustering

Huan Wang

***Abstract*—A novel clustering method is presented, which utilizes the Hopfield network as a way to optimize the clustering objective. First we formulate the clustering problem as a discrete quadratic optimization process. The interrelations among sample points are represented as a graph, and the clustering problem is interpreted as sub-matrix selection and rearrangement. Then we regard the optimization of the clustering objective as energy minimization, and we construct a Hopfield net with the same energy function as the clustering objective. The energy function is minimized using the Hopfield network. We prove that the optimal solution for clustering is a local minimum of the energy function, and analyze the relations between the convergence points of the associate Hopfield network and the desired clustering solution.**

***Index Terms*—Neural Network, Hopfield Net, Clustering, Image Segmentation, K-means, Graph Construction, Energy Minimization.**

I. Introduction

The venerable machine learning technique of clustering has been of active concern for decades. It is undergoing fast development with the new demands in computer vision, image processing, and machine learning. Fast, approximate algorithms, such as those based on spectral graph theory[8][5][11], manifold clustering[9][4], kernel k-means[10][3], have been proposed.

K-means [10] is perhaps the most widely used clustering algorithms. Its goal is to minimize the within-cluster distance. Instead of comparing samples pairwise, a representative sample for each cluster, i.e., the cluster center, is produced and the distance between samples and their cluster center, is minimized.

One drawback of k-means is that it can not guarantee the global optimality and the clustering results are thus subject to change with different initializations. The past ten years has seen the emergence of new clustering algorithms called spectral clustering. It arose [8] for a normalized cut of image pixels. Unlike the k-means algorithm that seeks a discrete but local optimization, the normalized cut algorithm transfers the problem to real values and a global optimum can be derived from eigen-decomposition of the associated graph Laplacian. The shortcoming of spectral clustering algorithms is that the global optimality is given in real domain, which means that the solution is still not globally optimal in discrete variable space.

Another problem for the k-means algorithm is when sample data are not favorably distributed, the clustering results can be quite bad. An example is when the high dimensional samples are distributed along a low dimensional manifold. Generally traditional k-means algorithm will fail. To overcome this limitation, the kernel k-means algorithm [3] was proposed. Instead of clustering on the Euclidean space, the samples are transformed to high dimensional feature space, and the k-means is conducted in the transformed space. The connection between the feature space and the original sample space is built upon the consistency of the inner-product, i.e., the Gram matrix. Essentially, the cluster centers can be derived from the associated pairwise kernel matrix, and thus the original sample vector is dispensable. By constructing different graphs, the spectral clustering algorithms can also accommodate the samples distributing over the manifold.

In this paper we build a Hopfield net for clustering. The connection coefficients between neurons are designed so that the energy function agrees with the proposed objective for clustering. We also discuss the relations between the proposed Hopfield net clustering algorithm, the k-means, and the kernel k-means algorithms. Experiments are carried out on both model examples and applications arising in image segmentation.

II. Hopfield Net and Energy Minimization

The Hopfield network updates the status of an associative memory according to the thresholds of different memory units and the interactions among them. The relationship between memory units can characterized by a graph G with adjacency matrix W. The graph is constructed so that its connections are consistent with the interconnections among memory units and the graph weights are set equal to the network coefficients.

The update rule of Hopfield net is:

$$Y = sgn(WY - b) \tag{1}$$

where $Y = [y_1, y_2, ..., y_n]'$ is the binary state vector with y_i as the status of memory unit i, $b = [b_1, b_2, ..., b_n]'$ is the threshold vector with b_i representing the threshold for the memory unit i. Both Y and b are n-by-1 column vectors, and the state $y_i = \pm 1$. W is the n-by-n adjacency matrix. Construction of W will be discussed in the following sections. All the graphs are assumed to be undirected and for such graphs, W is symmetric.

It can be shown that the iterative update procedure of Hopfield net will ultimately converge to a local minimum of the energy function:

$$E(Y) = -1/2Y^T WY + b^T Y \tag{2}$$

where the diagonal elements of W are zeros, i.e., the vertices are not self-connected.

III. Clustering and Binary Quadratic Optimization

Data clustering can be viewed as a mapping from the data vector space to the sample labels. Since the labels are

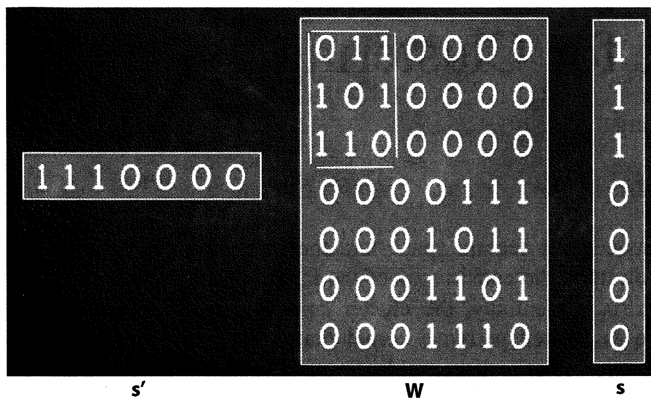


Fig. 1. Using selector vector to select different sub-blocks of a matrix. The matrix W is left multiplied and right multiplied with a selector vector s, and the sum of the elements within the selected sub-block is calculated.

categorical, we may represent them using integers. For two-way clustering, in particular, the labels can be expressed by $+1$ and -1, which is consistent with the status of memory units in the Hopfield net.

A. Cluster Association and the Adjacency Matrix

A large number of criteria have been proposed for clustering, such as graph cut minimization proposed by graph min-cut algorithms, normalized graph cut minimization proposed by normalized cut algorithms, information maximization proposed by information-based algorithms. We adopt the association maximization criterion as the clustering objective for our algorithm.

The intra-cluster association is taken to be the sum of all the intra-cluster weights (coefficients).

$$Asso(W) = \sum_{i,j} w_{ij} \tag{3}$$

B. Sub-matrix Selection by Matrix Multiplication

If we denote the labels of the two-way clusters as $+1$ and -1, then the intra-cluster association defined in the previous section can be calculated in matrix form as:

$$\begin{aligned} &Asso(W) \\ =&(e+Y)^TW(e+Y)/4 + (e-Y)^TW(e-Y)/4 \\ =&(e^TWe + Y^TWe + e^TWY + Y^TWY + e^TWe \\ &- Y^TWe - e^TWY + Y^TWY)/4 \\ =&(e^TWe + Y^TWY)/2, \end{aligned} \tag{4}$$

where Y is the cluster indicator vector with elements $+1$ and -1, and e is a vector of ones.

$(e+Y)/2$ is the selector vector for the first cluster. It has elements of 1 for those points within the corresponding cluster and 0 for the other points. Similarly $(e-Y)/2$ is the selector vector for the second cluster. Consequently, the terms $(e+Y)^TW(e+Y)/4$ and $(e-Y)^TW(e-Y)/4$ are the intra-cluster association of the corresponding clusters. Figure (1) shows the way to use a selector vector s for a sub-block sum of the matrix W.

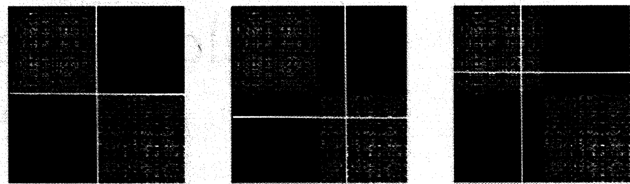


Fig. 2. The desired solution is a local maximum of Association. (Not the global maximum). From left to right, desired solution, initialization a, initialization b. Either the initialization a or b will lead to a desired solution.

Maximizing the association $Asso(W)$ is equivalent to maximizing Y^TWY, according to Equation (4). This objective agrees with the the energy function minimization of Hopfield net $E(Y)$ (Eqation (2)).

IV. Graph Construction

Let $G = (V, E)$ denote the graph with vertex set V and edge set E constructed with the data. We shall restrict our attention to undirected graphs. Edges E reflect the neighborhood relations along the manifold data, which can be defined in terms of k-*nearest* neighbors or an ϵ-*ball* distance criterion. Choices for those non-negative weights on the corresponding edges come from the heat kernel [1] or the inverse of feature distances [2], i.e.,

$$w_{ij} = e^{\frac{-\|x_i - x_j\|^2}{t}} \quad \text{or} \quad w_{ij} = \|x_i - x_j\|^{-1},$$

where $t \in \mathbb{R}$ is the parameter for the heat kernel. Another approach is to solve a least-square problem to minimize the reconstruction error to specify the weights w_{ij}[6]:

$$w_{ij} = \arg\min_{w_{ij}} \|x_i - \sum_j w_{ij} x_j\|^2,$$
$$s.t. \sum_j w_{ij} = 1, w_{ij} \geq 0$$

V. Update Rule and Selection of the Threshold

We select the objective of association maximization clustering as the energy function, and construct a corresponding Hopfield net with coefficient matrix W_{cof} equal to the graph adjacency matrix W. If the threshold vector b is set to zero, the energy function of the constructed Hopfield net will be exactly the same as the clustering objective. Then following the update rule of Hopfield net, we can reach a local minimum of the Energy function $E(Y)$:

$$Y(n+1) = sgn(WY(n)) \tag{5}$$

Figure (2) shows two initializations and the desired local minimum.

VI. Relations to K-Means Algorithm

For each cluster i, denote the cluster center as c_i, the samples as x_i and the sample labels as y_i^l. Then the distance between samples x_i and their cluster center c_i is

$$min_{y_i^l, c_i} \sum_{c_j} \sum_{y_i \in c_j} \|x_i - c_j\|^2 \quad (6)$$

Assume now we have the cluster centers c_i derived from the previous steps, then,

$$y_i^l = argmin_j \|x_i - c_j\|^2, \quad (7)$$

Equation (7) reaches its minimum when y_i^l is derived using the nearest neighbor method w.r.t. c_j. It is not hard to see this conclusion since if one sample is not assigned to its nearest cluster, the objective $\|x_i - c_j\|^2$ will be larger compared to the nearest neighbor assignment.

Next, if we are given the cluster labels y_i^l, the cluster centers to be derived are:

$$c_j = argmin_{c_j} \sum_{y_i^l \in c_j} \|x_i - c_j\|^2, \quad (8)$$

which can be minimized when $c_j = \sum_{y_i^l \in c_j} \sqrt{\|x_i - c_j\|^2} / S_i$, where S is the number of points in cluster i.

From equation (7) and (8), the objective $\sum_{y_i^l \in c_j} \|x_i - c_j\|^2$ always declines. However, the K-means algorithm can not guarantee the global optimality[10].

The centers c_j can be represented in matrix form as:

$$c_j = \Phi_j e / s_j, \quad (9)$$

where $\Phi_j = \{x_i | y_i = j\}$, and s_j is the number of point in cluster j. Denote the sum of within cluster distance for cluster j as:

$$d_j = \sum_{y_j = j} \|x_j - c_j\|^2 \quad (10)$$

$$= \sum_{y_j = j} \|x_j - \Phi_j e / s_j\|^2, \quad (11)$$

$$= \|\Phi_j - \Phi_j e e^T / s_j\|_F^2 \quad (12)$$

$$= \|\Phi_j (I - e e^T / s_j)\|_F^2, \quad (13)$$

where I is the identity matrix. Note here $I - ee^T/s_j$ is the graph Laplacian L^{kmean} of the graph specified by the adjacency matrix $W^{kmean} = ee^T/s_j$, and L^{kmean} is orthogonal projection, i.e., $L^{kmeanT} L^{kmean} = L^{kmean}$. Then

$$d_j = trace(\Phi_j L^{kmean} \Phi_j^T). \quad (14)$$

According to the theory of the graph Laplacian [1],

$$trace(\Phi_j L^{kmean} \Phi_j^T) = \sum_{uv} \|x_u^j - x_v^j\|^2 / (2 s_j), \quad (15)$$

where x_u^j is a sample of cluster j.

Thus the objective of the k-means algorithm can be formulated as:

$$d = \sum_j d_j \quad (16)$$

$$= \sum_j \sum_{uv} \|x_u^j - x_v^j\|^2 / (2 s_j), \quad (17)$$

as in matrix formulation:

$$argmin_{Y^l} trace(Y^{lT} M^{norm} Y^l) = Asso(M^{norm}), \quad (18)$$

where $Y^l = [y^{cl_1}, y^{cl_2}, ..., y^{cl_k}]$ is the label matrix with $y^{cl_i} = [y_1^{cl_i}, y_2^{cl_i}, ..., y_n^{cl_i}]$ being the sample labels of cluster cl_i ($y_i^{cl_i} \in \{0, 1\}$), and M^{norm} is the normalized distance matrix with elements equal to the distance divided by its corresponding s_j.

A similar analysis can be done for the kernel k-means. The only difference is, instead of the normalized negative distance matrix M^{norm}, we will use the 'kernel trick' [7] to derive the normalized distance matrix M^{norm} in the transformed Hilbert space.

From objective (17) we see that if we replace the graph adjacency matrix W with $-M^{norm}$, the only difference between the objectives of the k-means and the Hopfield net is that the k-means has a normalization term s_j. Unfortunately the objective (17) is time variant, and after the normalization, M^{norm} is no longer symmetric, which means traditional Hopfield net can not work for the k-means optimization. Still we notice that there exists an exact correspondence between the update rule of our proposed algorithm and the k-means. Since $y_i \in \{+1, -1\}$, the update rule (5) essentially compares for each sample two distances from sample to the two clusters, i.e., the $+1$ cluster minus the -1 cluster. If the $+1$ cluster distance is larger, after *sign* function, the sample will be assigned as $+1$. Remember here we have a negative sign, the sample will finally be assigned to the cluster with a smaller distance.

The connection between the pairwise distance and the distance to the cluster mean is

$$\|x_n - c_j\|^2 = \|x_n - \sum_{y_i \in c_j} x_i / s_j\|^2 = \|\sum_{y_i \in c_j} (x_n - x_i)\|^2 / s_j^2$$

$$= \sum_{y_u, y_v \in c_j} (x_n - x_u)(x_n - x_v) / s_j^2 \quad (19)$$

Thus if we modify the graph W so that

$$w_{ij}^m = - \sum_{y_j, y_k \in c_j} (x_i - x_j)(x_i - x_k) / s_j^2, \quad (20)$$

Then the two distances from each sample to the two cluster centers compared in the update rule will be the same as the k-means algorithm does. And the algorithm will perform exactly in the same manner as the k-means method. Note at this time the algorithm is not the traditional Hopfield net anymore, since W is both asymmetric and time variant.

VII. Convergence Analysis

In this section we will give an intuitive analysis for the algorithmic convergence. As demonstrated in Figure (2), no matter the initialization a or b, as long as it is not too far from the desired local optimum, the state vector will move towards the desired solution (Figure (3)). While when the initialization is too far away, the iterative procedure will converge to the global optimum with a trivial solution (Figure (4)).

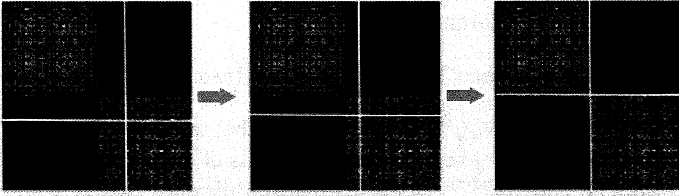


Fig. 3. A successful initialization and the desired convergence. Since the initialization is not far from the desired local optimum, minimizing the energy function will ultimately push the neuron status to the desired solution.

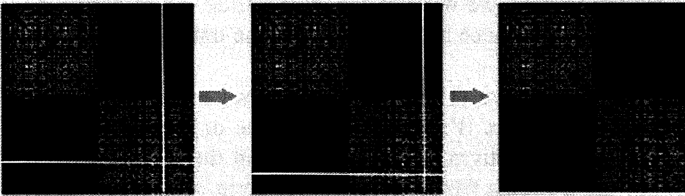
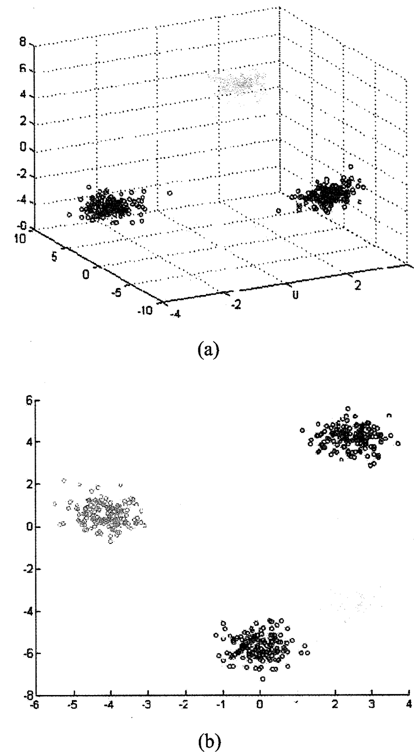


Fig. 4. When the initialization is not proper, it will converge to a trivial solution. The minimization of the energy function in this case only produce a vector of ones. Although it is the global optimum, the solution is not what we want.



(a)

(b)

Fig. 5. Multi-way clustering on toy examples. (a) three-way clustering. (b) four-way clustering. The samples are generated by the Gaussian mixture model with mixture number of 3 and 4 correspondingly. The standard deviation is set 0.5, and for each cluster 150 samples are generated.

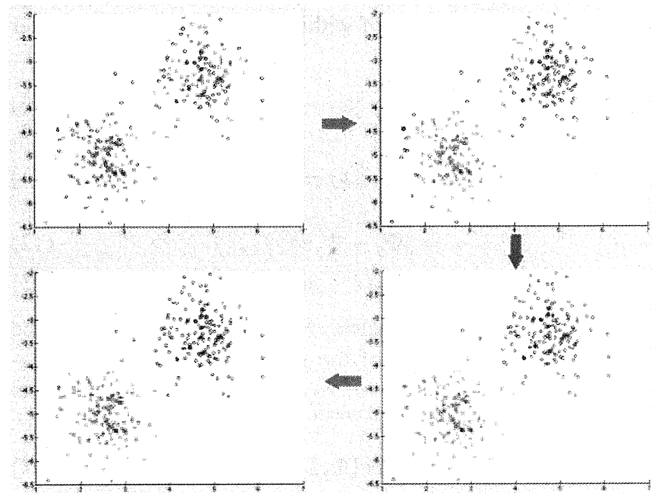


Fig. 6. A successful clustering procedure. The iterative process converges after 4 steps. Cluster labels are expressed using different colors.

VIII. Experiments

A. Demonstration of the Convergence

In this sub-section we show the convergence process of the two-way clustering problem on a model example. The samples are generated from a 2 dimensional Gaussian mixtures with mixture number equal 2. The standard deviation σ is set 0.5, and for each mixture 150 samples are generated. Figure (6) shows a successful convergence process. After 3 iterations the process converges to the desired solution. Figure (7) shows a convergence process to a trivial solution when the initialization is not proper. A trick is to initialize as the k-means algorithm does: we first select cluster centers randomly and get the initial sample labels using the nearest neighbor method. From the experiments we find this initialization is more stable compared to the uniform random label generator.

B. Multi-way Clustering Results on Toy Examples

An extension of the two-way clustering problem stated above is the multi-way clustering problem. This can be simply done by iteratively conducting the two-way bi-clustering algorithm to the previously derived clusters. If an odd number of clusters is desired, e.g., 3 clusters, we simply pick out the cluster with a largest sample number to feed the next iteration, and then the cluster with the second largest sample number, and so on. The clustering results are demonstrated in Figure (5).

C. Image Segmentation

In this sub-section we conduct data clustering on image pixels, that is, image segmentation. The color images are first converted to gray level images. Then the graph is constructed following [8]. The Hopfield net is constructed so that the coefficient matrix equal to the graph weight matrix W. Colored images are first converted into gray-level images during the preprocessing stage. Then they are feed to the clustering algorithm to get the label of each pixel. The clustering results are compared with those of the k-means algorithm in Figure (8), from which we can see that both algorithms produce sound clustering results which characterize the main structures in the

Fig. 8. Image segmentation results. The first column is the original images, the second column is the clustering results derived by Hopfield Net, and the third column is the k-means clustering results. Edges between the two segmentations are indicated by red lines. During the preprocessing stage, the RGB color image is converted into gray-level image.

image. The segment edges produced by the Hopfiled net are generally smoother when compared to the traditional k-means algorithm.

IX. Conclusions and Future Work

We present a method that using the Hopfield net as a tool for clustering. The coefficients, i.e., the graph weights W, between neurons are designed so that the energy function agrees with the clustering objective, i.e., maximize the within cluster weights or minimize the within cluster distance. Also, we discuss the relations between our proposed algorithm, the k-means and kernel k-means algorithms. We plan to build the connections with the spectral clustering algorithms in future work.

References

[1] M. Belkin and P. Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural Comput.*, 2003.
[2] C. Cortes and M. Mohri. On transductive regression. In *NIPS*. 2007.
[3] I. S. Dhillon, Y. Guan, and B. Kulis. Kernel k-means: spectral clustering and normalized cuts. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 551–556. ACM Press, 2004.
[4] R. Haralick and R. Harpaz. Linear manifold clustering in high dimensional spaces by stochastic search. *Pattern Recogn.*, 40(10):2672–2684, 2007.

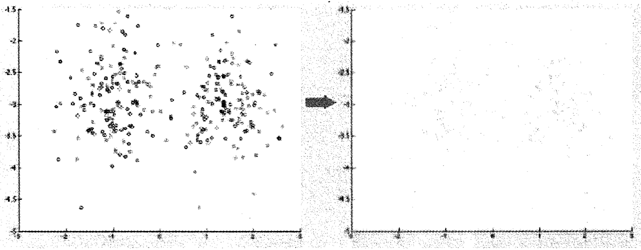


Fig. 7. Bad initialization and trivial solution. In this case the algorithm produced a vector of ones, thus all the samples are labeled with the same color.

[5] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems 14*, pages 849–856. MIT Press, 2001.

[6] S. T. Roweis and L. K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 2000.

[7] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, USA, 2001.

[8] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[9] R. Souvenir and R. Pless. Manifold clustering. In *In ICCV*, pages 648–653, 2005.

[10] A. B. Y. Linde and R. Gray. An algorithm for vector quantizer design. *IEEE Transactions on Communications*, 1980.

[11] S. X. Yu and J. Shi. Multiclass spectral clustering. In *In International Conference on Computer Vision*, pages 313–319, 2003.

Self-Organized Online Network Traffic Forecasting using Neural Networks

Ye Wang

Abstract—**Self-organizing neural network system for online forecasting (internet) network traffic is introduced. The foundation of the system is that neural networks can identify and quantify the deep relationship and complicated interaction inside network traffic pattern. Based on this knowledge, neural networks are then able to forecast future network traffic. A Self Organizing Map and a multilayer feedforward predictor are employed and integrated by the network traffic forecasting system. Evaluation by system simulation has demonstrated the forecasting accuracy (90-percentile error is less than 3.5% of device bandwidth) brought by the self-organization of network traffic and online study of the intrinsic relationship inside traffic.**

Index Terms—**Neural networks, Self-Organizing Map, Computer networks, Traffic forecasting**

I. Introduction

Accurate and efficient real-time network traffic forecasting is of particular interest for internet design, operation and application, e.g., topology plan, bandwidth provision, traffic engineering, active traffic control, intrusion detection and reaction, service differentiation, self-adaptive application design, etc. However, due to self-similarity and long-range dependence of internet traffic[1], [2], [3], [4], [5], it is inherently difficult to have a simple rigorous algorithm that can accurately predict future network traffic levels.

There are many proposed heuristics and techniques in the field of network traffic forecasting (e.g., [6], [7], [8]), some of which are based on neural networks (e.g., [9], [10], [11]). Each contribution has its own insight and advantage in its particular usage context. But generally, little has been considered in previous works that the unpredictability of network traffic is caused by both *higher-layer internet user behavior* and *lower-layer protocol dependencies*.

Our novel approach integrates a self-organizing map and a feedforward multilayer predictor to identify and quantify the deep *relationship* among a substantial number of IP packets. Based on that, our system forecasts network traffic level in the near future, taking cognizance of the complicated *interaction* inside the current traffic pattern.

We present the neural network system architecture and the design rationale in Section II. After that, we detail the SOM and the multilayer predictor in Section III and IV, respectively. Evaluation is made in Section V, where we will also discuss issues of implementing this system in real network devices.

II. System Overview

The network traffic forecasting problem we address is: given the latest history (e.g., 100ms, or one time slot) of incoming packets (possibly header only) captured by a network device (e.g., router, switch), what is the predicted traffic level (e.g., device bandwidth utilization) in the next time slot?

Studies[1], [2], [3], [4] have shown that internet traffic is "bursty" so that the network traffic level is a random process with self-similarity and long-range dependence. This is the reason why traditional prediction filtering (e.g., Kalman Filter[12]), based on ergodicity and short-range dependence, does not usually work well for network traffic forecasting.

In addition to the statistical-sharing design of the internet, we argue that the unpredictability of network traffic is results for two indirect reasons:

- higher-layer internet user behavior: internet user behavior is highly unpredictable in terms of the traffic a user generates on the network. Sometimes they exhibit very similar behavior, such as flush crowd on `www.cnn.com` during the presidential election event. Most of the time they behave totally independently. For example, John is watching `youtube` and Mike is sending worm-infected emails. Moreover, one user can generate a great amount of traffic, while most of the users are light-loaded.
- lower-layer protocol dependencies: Almost every popular internet application runs on more than one protocol. As an example, an email application may require DNS, SMTP, POP3, IMAP, HTTP, and even FTP. Also, network operators have to run many other protocols to support applications. Such protocols include routing protocols like IS-IS, IGRP, BGP, and management essentials such as ICMP and SNMP. Complicated network traffic patterns are caused by the interactions among these protocols. Furthermore, even for a single protocol, the IP packets generated can be inter-dependent, such as when a TCP connection is used.

Even if these two reasons can be identified and quantified, designing simple rigorous algorithms to capture them in network traffic measurement and forecasting is particularly challenging because of the associated exponential complexity ("curse of dimensionality").

For such reasons, neural network techniques can be helpful. We build up a neural network system that self-organizes the traffic identifying deep intrinsic relationships and then forecasts the future network traffic, taking cognizance of the complicated traffic interaction.

The system architecture is illustrated in Fig.1. It consists of two major components:

- Traffic self-organization based on SOM: self-organize the

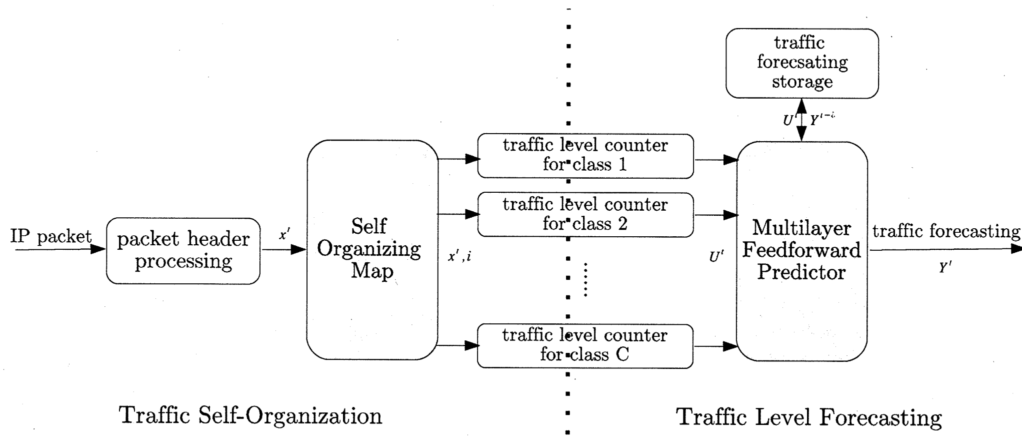


Fig. 1. System Architecture

packets into adapted classes according to packet headers, and count traffic level of each class;

- Traffic level forecasting based on feedforward multilayer predictor: predict the traffic level of each class in the next time slot, and sum up the total traffic level.

In the following two sections, we will present the detail design of each component.

III. TRAFFIC SELF-ORGANIZATION

Network traffic self-organization is the first step for our neural network system to learn the intrinsic relationship among the network traffic.

A. Component Design

During each time slot, incoming packets are classified into a fixed number of classes according to packet headers. The classification is self-organized in the sense of that a class is (re)identified by the density of similar packet headers. The traffic level of each class is then counted.

The function of this component is not purely to classify network traffic flows, but to self-adapt the classes according to the current traffic pattern. Therefore, the self organizing map (SOM)[13] algorithm is employed in this component.

SOM uses competitive learning to process the distribution of the input data in a high-dimensional space and adapt each neuron to map this distribution into a low-dimensional space. We argue that SOM is good at learning the complicated relationship among a great number of IP packets. Though the higher-layer user behavior and lower-layer protocol dependencies are complicated, they are eventually encoded in the packet header, and so can be studied employing the SOM.

B. Traffic Self-Organization based on SOM

At time t, the current time slot T^t is a time range $(t - \delta, t)$, where δ is taken to be 100ms. Each incoming packet is classified by the SOM into a particular class i. A counter, following the SOM, counts the number of packets N_i^t in each class i during time slot T^t. N^t, as the current traffic level, is used as the input of the next system component, traffic level forecasting.

The SOM employed in network traffic self-organization is responsible for recognizing the network traffic pattern and identifying the traffic classes based on the pattern.

The input to the map is a sequence of 5-dimensional vectors $x \in [0, 1]^5$, resulting from compression of the traffic information extracted from a packet header. We select six fields from the packet header: timestamp, source and destination IP address, source and destination ports, and transport protocol. Timestamp is used by the counters to align the timeslot, while the other five fields are normalized and reconstructed as the input x to the SOM.

- Source IP Address: $SrcIP$, 32-bit integer, the source of the IP packet, $x_1 = SrcIP/2^{32}$;
- Destination IP Address: $DstIP$, 32-bit integer, the destination of the packet, $x_2 = DstIP/2^{32}$;
- Source Port: $SrcPort$, 16-bit integer, the source port of the transport protocol, $x_3 = SrcPort/2^{16}$;
- Destination Port: $DstPort$, 16-bit integer, the destination port of the transport protocol, $x_4 = DstPort/2^{16}$;
- Protocol: $Protocol$, TCP, UDP, ICMP, etc. We distinguish TCP from other protocols. If $Protocol = TCP$ then $x_5 = 1$ else $x_5 = 0$.

The selection of packet fields and the construction of input x is an interesting topic but out of the scope of this paper. For example, considering the pattern of packet payload may be helpful for self organization. We use traditional flow-related packet header fields in our system only to demonstrate our basic idea: neural networks can help identify complicated relationship inside network traffic.

The number of neurons, equal to the number of traffic classes, is $C = 20$. The output is i, the index of the winning neuron, or the class of the packet. The structure, learning algorithm, and parameters of the SOM are listed in Table I.

TABLE I
SOM PARAMETERS

Number of Neurons	20
Topology	random
Initial Weights	uniform
Distance Function	Euclidean distance
Learning Rule	Kohonen algorithm
Learning Epochs	100

IV. Traffic Level Forecasting

After self-organization, the next step is to study the relationship among the traffic classes and, most importantly, what impact the relationships have on the future traffic level of each class.

A. Component Design

The network traffic is self-organized prior to component design. Given current traffic level of each class, we use a multilayer feedforward neural network to predict the traffic level of each class a time slot later.

The simplest neural network that can be used as a predictor is the memoryless feedforward network[13]. As mentioned, adaptive filters based on feedback do not fundamentally improve the accuracy of network traffic forecasting. Therefore, we employ a simple 2-layer feedforward neural network in this component. We choose the efficient reduced memory Levenberg-Marquardt algorithm[14] as the backpropagation training rule of the predictor.

Different from the common offline training of multilayer feedforward network, this predictor is trained adaptively in an online manner. The predicted traffic level during the latest time slot is stored in this component and will be used for training in the next time slot. The underlying reason is that the network traffic classes are self-(re)organized and thus the relationship among the different classes of traffic is not fixed. Through online adaptive learning, we envision the synaptic weights in the predictor can always encode the complicated relationship and interaction inside the latest network traffic pattern.

B. Multilayer Feedforward Predictor

The parameters of this neural network are listed in Table II. The input of the predictor is a $C(=20)$-dimensional vector $U^t = \{U_i^t\}_{i=1..C}$, where $U_i^t = N_i^t/B$ is the normalized ($U^t \in [0,1]^C$) traffic level of class i during time slot T^t. B is a normalizing scalar, e.g., the device bandwidth. The output is the predicted traffic level $Y^t = \{Y_i^t\}_{i=1..C}$, where Y_i^t denotes the future traffic level of class i at time $t+\delta$. The overall traffic level u^t can be denoted as the sum of each row of U^t, and the overall prediction y^t is then the sum of each row of Y^t.

Note that U^t as the real traffic level is also used as the desired output $D^{t-\delta} = U^t$. At time t, $D^{t-\delta}$ and the stored $Y^{t-\delta}$ are compared, and the error between the prediction and the real traffic level is backpropagated in the neural network so that online adaptive training is achieved.

TABLE II
MULTILAYER FEEDFORWARD PREDICTOR PARAMETERS

Number of Layers	2
Number of Neurons	20 per layer
Initial Weights	random
Learning Rule	Backpropagation with Reduced Memory Levenberg-Marquardt
Learning Epochs	100

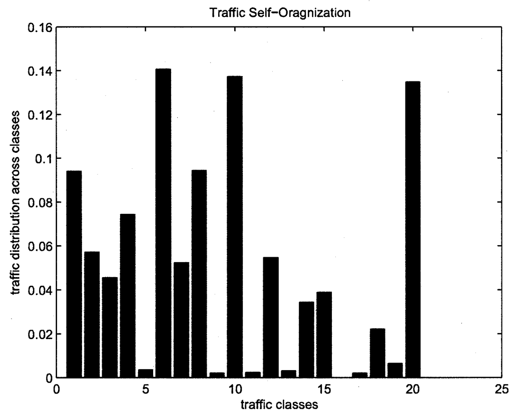


Fig. 2. Traffic Self-Organization (18:00:00-18:00:01)

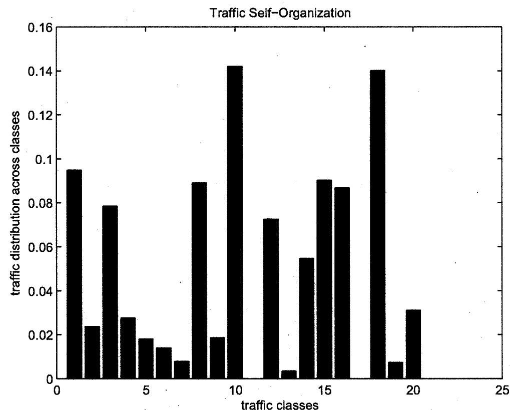


Fig. 3. Traffic Self-Organization (18:00:02-18:00:03)

V. Evaluation

We evaluate the accuracy of the system, the impact of each component, and the performance sensitivity to missing data through simulation. The neural network system is implemented based on MATLAB Neural Network Toolbox[15]. The network traffic data used in the simulation is the CAIDA[16] packet trace captured during 18:00-19:00 on Jan 9, 2007 (UGT) on an OC12 link at the AMPATH Internet Exchange[17].

A. Traffic Self-Organization

We extract two one-second-long packet traces (1. from 18:00:00 to 18:00:01, 3082 IP packets; 2. from 18:00:02 to

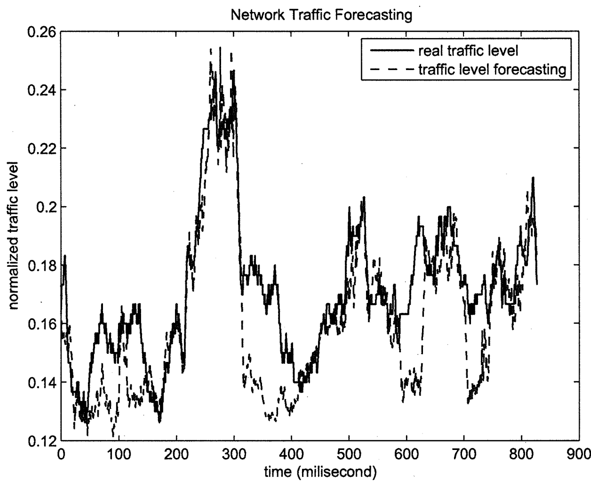


Fig. 4. Traffic Level Forecasting (by our system)

18:00:03, 3825 packets) and use the traffic self-organization component to classify the traffic continuously. The identified classes and the traffic distribution across the classes for these two different time range are plotted in Fig 2 and Fig 3.

The results of traffic self-organization has demonstrated the intrinsic complexity and the non-uniform pattern of the network traffic. It has also been observed that the traffic pattern evolves quickly in time, and so the traffic classification can vary.

B. Traffic Forecasting Accuracy

The traffic level forecasting of the same packet trace segment as in the previous subsection is demonstrated in Fig 4.

The predicted traffic level is observed to be close to the real traffic level, though we see one interesting inconsistency comparing the two curves: when the traffic level tends to diminish, the prediction falls much faster and deeper than the real traffic level. A hypothetical reason is that the system is sensitive to bursty traffic. The traffic classes may have to be reorganized when a burst gets offline, so it requires relearning for the predictor. Fortunately, a network device is likely to be more interested in peak traffic levels, so it might be tolerable if the forecasting is less accurate only under light load.

To quantify the traffic forecasting accuracy, we define the forecasting error to be $e^t = |u^{t+\delta} - y^t|$ and plot the cumulative distribution of e^t in Figure 5. We do not renormalize e^t by $u^{t+\delta}$ since we are less concerned with the relative error under low traffic level $u^{t+\delta}$. Essentially, e^t is already normalized by the network device bandwidth.

It is shown that the maximal forecasting error is about 0.06 ($e^t \in [0, 1]$). Over 90% of the time the traffic level forecasting has no more than a 0.035 error. The average forecasting error is 0.0134, and the median is 0.0087.

We also tested the performance of our system on traffic traces in other time ranges, and the forecasting accuracy is similar to the case presented.

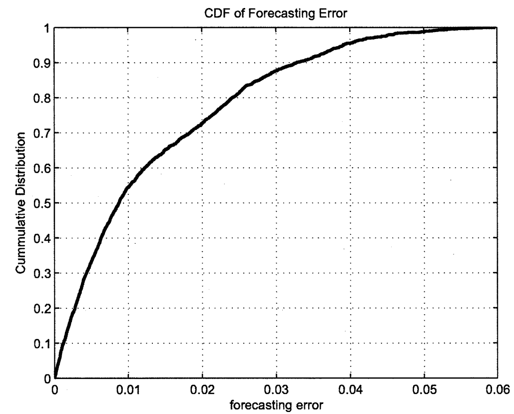


Fig. 5. Traffic Level Forecasting Error

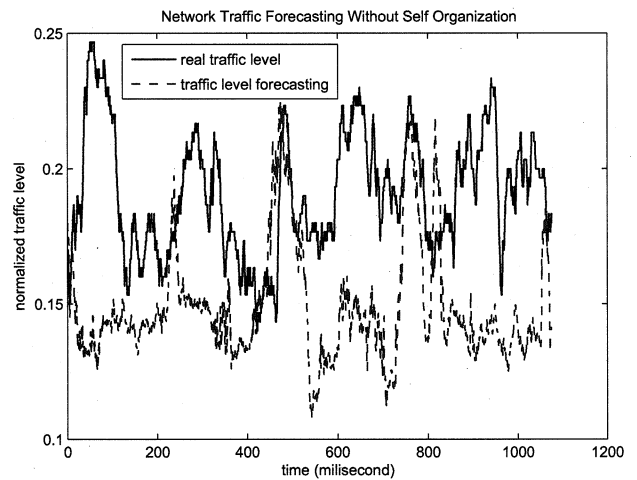


Fig. 6. Traffic Level Forecasting Without Self-Organization

C. Impact of Traffic Self-Organization

One question to the proposed system is: if the traffic is classified only according to the historical data, will the forecasting still be accurate? To answer this question, we train the system using the one-second packet trace (from 18:00:00 to 18:00:01), fix the neuron weights of the SOM, and test the forecasting using another one-second (from 18:00:02 to 18:00:03) with the fixed-weight SOM.

The forecasting result is plotted in Fig 6, and the forecasting error in Fig 7. Though the forecasting can sometimes match the real traffic, we observe significant drop of the forecasting accuracy. Compared with the previous results, the 90-percentile forecasting error increases to over 0.08.

This is not surprising because the network traffic pattern is always evolving. Without self-(re)organization, the impact of traffic pattern changes would not be learned by the neural network. Therefore, forecasting based on fixed traffic classification is no working well.

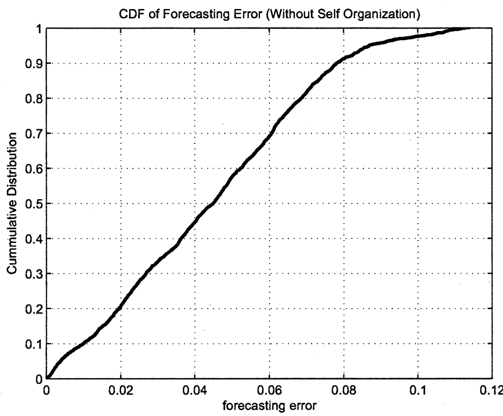


Fig. 7. Traffic Level Forecasting Error Without Self-Organization

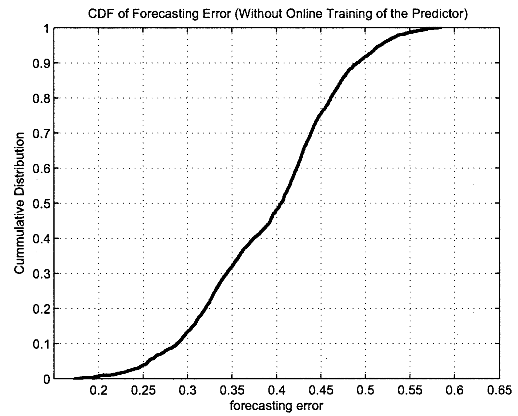


Fig. 9. Traffic Level Forecasting Error Without Adaptive Training of the Predictor

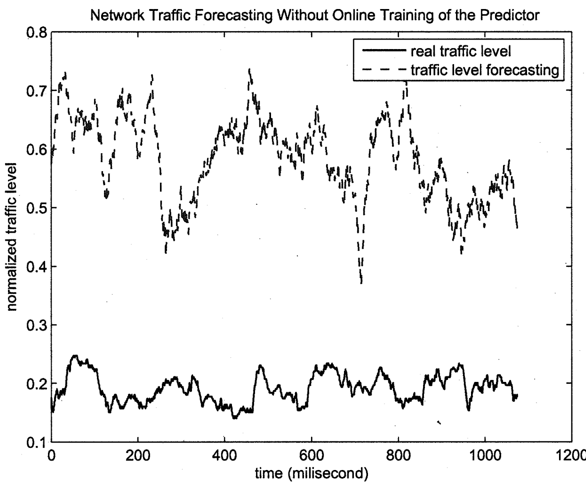


Fig. 8. Traffic Level Forecasting Without Adaptive Training of the Predictor

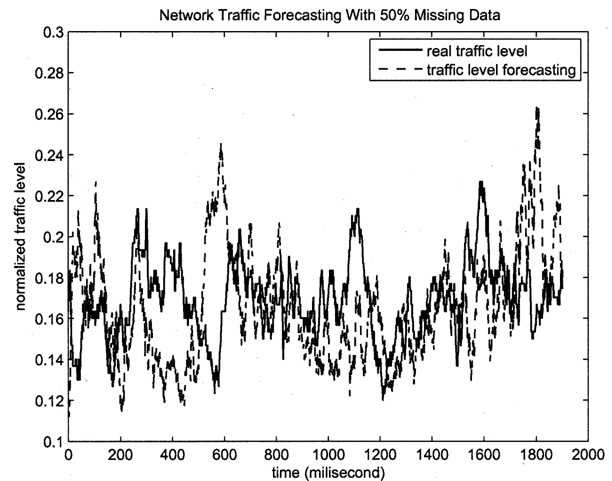


Fig. 10. Traffic Level Forecasting With 50% Missing Data

D. Impact of Adaptive Online Training

Another issue we would like to evaluate is how adaptive online training of the multilayer feedforward predictor help the accuracy of forecasting. Similar as evaluating the SOM, we train the system using the one-second packet trace (from 18:00:00 to 18:00:01), fix the neuron weights of the predictor, and test the forecasting using the next one-second (from 18:00:02 to 18:00:03) with the weights-fixed predictor.

The forecasting result is plotted in Fig 8, and the forecasting error in Fig 9. The traffic level forecasting is completely different from the real traffic level, and the average forecasting error reaches as high as 0.5. The results indicate that the self-reorganization must be learned by the multilayer feedforward network dynamically, or the predictor would not be able to do meaningful forecasting.

E. Impact of Missing Data

Real network devices are not usually able to capture all packets. The reasons can be underlying physical hardware failure, internal software error, or most likely, bandwidth overload. Therefore, only sampled packets can be used for training neural networks.

We want to understand whether randomly missing data in system training would increase forecasting error, so we train the system using a two-second packet trace (from 18:00:00 to 18:00:02) but assuming 50% of the packets randomly dropped. The forecasting accuracy with missing data is shown in Fig 10 and Fig 11. The error is almost twice as high as when the system is trained with all data.

We also consider the case where missing data is caused by data sampling. Across the two-second packet trace (from 18:00:00 to 18:00:02) every other packet is sampled. Thus 50% of the traffic data is still learned by the neural networks. The traffic level forecasting accuracy is tested and the result is similar as under random missing data.

The degraded forecasting accuracy has shown that missing data in neural network system training due to packet drop or sampling may lead to significant loss of important information inside the traffic pattern. Therefore, the usefulness of our system may rely on a real-time packet capture technique that

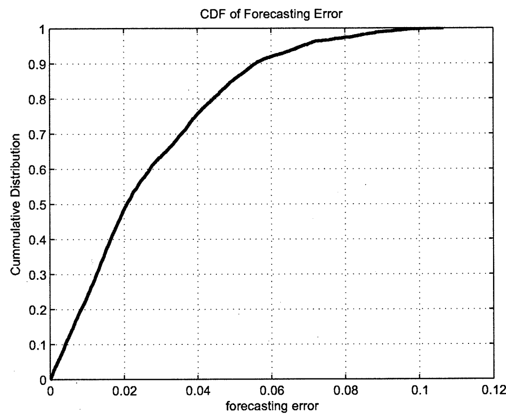


Fig. 11. Traffic Level Forecasting Error With 50% Missing Data

TABLE III
SYSTEM COMPLEXITY

Number of Neurons	60
Number of Synopsis	≤ 500
Training Time (for 100ms trace)	$<$ 100ms
Response Time (for 100ms trace)	$<$ 1ms

can guarantee minimal data loss.

F. System Complexity

To support high scalability and efficiency, real network devices (routers, switches) usually employ simple hardware and software design. Thus, integration of neural networks into real network device implementation requires low complexity of the neural networks in terms of small memory space consumption and short training time.

We investigate the complexity of the whole system, and list the parameters in Table III. The training time consumption is estimated by MATLAB simulation running on a commodity computer. The numbers suggest that it should be feasible to port the neural networks into online network devices.

VI. CONCLUSION

This paper builds a self-organized network traffic forecasting system using neural networks. Through self-organizing traffic into adapted classes and dynamically learning the deep relationship and complicated interaction among traffic classes, the system achieves accurate and efficient network traffic forecasting.

ACKNOWLEDGMENT

The author would like to thank Prof. Willard Miranker and Jaeoh Woo for the benefitial initial discussions and revision suggestions.

REFERENCES

[1] K. Park and W. Willinger, *Self-similar network traffic and performance evaluation.* John Wiley & Sons, 2002.

[2] M. Grossglauser and J.-C. Bolot, "On the relevance of long-range dependence in network traffic," *IEEE/ACM Transactions on Networking*, vol. 7, pp. 629–640, 1999.

[3] M. E. Crovella and A. Bestavros, "Self-similarity in World Wide Web traffic: evidence and possiblecauses," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 835–846, 1997.

[4] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similarity nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking*, vol. 2, pp. 1–15, 1994.

[5] J. Beran, *Statistics for long-memory processes, monographs on statistics and applied probability*. New York, NY: Chapman and Hall, 1994.

[6] H. Feng and Y. Shu, "Study on network traffic prediction techniques," in *WiCOM*, 2005.

[7] J. Ilow, "Forecasting network traffic using farima models with heavy tailedinnovations," in *ICASSP*, 2000.

[8] A. Sang and S. Li, "A predictability analysis of network traffic," in *Proceedings of IEEE INFOCOM '00*, Tel Aviv, Israel, Mar. 2000. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/RecentCon.htm?punumber=6725

[9] A. Khotanzad and N. Sadek, "Multi-scale high-speed network traffic prediction using combination of neural networks," in *Proceedings of International Joint Conference on Neural Networks*, 2003.

[10] G. Cheng, J. Gong, and W. Ding, "Nonlinear-periodical network traffic behavioral forecast based on seasonal neural network model," in *ICCCAS*, 2004.

[11] D. Park, "Prediction of network traffic using multiscale- bilinear recurrent neural network with adaptive learning," in *ICIC*, 2008, pp. 525–532.

[12] G. Welch and G. Bishop, "An introduction to the kalman filter," Tech. Rep. TR95-041, 1995.

[13] S. Haykin, *Neural Networks: A Comprehensive Foundation.* Pearson Education, Second Edition, 1999.

[14] J. More, "The Levenberg-Marquadt algorithm: implementation and theory," in *Proceedings of the Biennial Conference of Numerical Analysis*, 1978.

[15] H. Demuth and M. Beale, "MATLAB neural network toolbox: user's guide," 1997.

[16] "Caida cooperative association for Internet data analysis." http://www.caida.org/.

[17] "Ampath International Exchange Point in Miami," http://www.ampath.fiu.edu/.