Hopfield Networks as Classifiers of Handwriting

Nicholas Hooper and Heather Mattie

***Abstract*—Hopfield networks are fully connected networks which contain multiple neurons, each of which is connected to the others via weights. A key feature of Hopfield networks is that they converge to a local minimum (a.k.a "lowest energy state"). Partial or corrupted memories can be restored to their correct state. In addition, inputs that are sufficiently close to a stored memory will converge to the desired memory. The choice of weights is key in ensuring the highest rates of accuracy. Hebbian learning optimized via gradient descent seems to be the best choice to optimize the weights while preventing overfitting of the neural network.**

***Index Terms*—Handwriting recognition, Hopfield networks, neural networks, neural network applications**

I. Introduction

ARTIFICAL neural networks are mathematical constructs that model certain aspects of biological neurons. Neural networks consist of one or more neurons which can exhibit complex behavior due to the nature of the connections between neurons. Neural networks can serve to store "memories" of data. These memories mimic biological systems in several ways.

1) They are associative. With associative memories, one can recall a memory through its association with other memories. In other words, if you associate *banana* with *yellow*, you can recall *banana* if you are given *yellow* and vice versa.
2) They are robust in handling error. In a neural network, memories can correct for a certain amount of error and still recall the correct memory.
3) Memories are distributed. In a neural network, each neuron plays a part in memory storage. This is opposed to traditional computing, where a memory is associated with a specific location on the storage device.

Hopfield networks are a type of neural network named for John Joseph Hopfield (an American Scientist). They are fully connected networks which contain multiple neurons, each of which is connected to the others via weights. A key feature of Hopfield networks is that they converge to a local minimum (a.k.a "lowest energy state"). This feature creates applications in two main areas: associative memories and optimization. With associative learning, the network can be trained to "remember" states which can be recalled if it is given only part of the state. Within optimization problems, as the network converges to its local minima, the neural networks' objective function can be optimized for a desired solution.

II. Background

A. A Single Neuron

We begin our discussion of Hopfield networks by discussing the structure of a single neuron. Our Hopfield network is constructed of many single neurons which are linked. A neuron's behavior is determined by three parts: the architecture, activity rule, and learning rule.

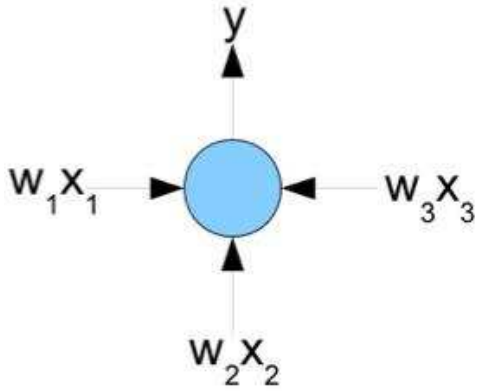

Fig. 1. A single neuron

The *architecture* describes the variables of a neuron and their relationship to on another. Each parameter (x_i) has a corresponding weight (w_i).

The *activity rule* describes the dynamics of a neuron, i.e. how each neuron changes in response to the parameters. It has two steps.

1) First we determine the activation of the neurons
$$a_i = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x} \tag{1}$$
2) The output y of a neuron is a function of the activation ($f(a)$). This is called the activity of the neuron. There are several possible activation functions, and the most popular include linear, logistic, hyperbolic tangent, and threshold functions.

The *learning rule* determines the neuron's weights (**w**), and its structure will depend on the type of neural network we are running. Often this involves the minimization of an objective, or error, function.

B. Hopfield Networks

In a neural network, the output from one neuron is the input into another. Their relationship falls into two categories: 1) feedforward networks and 2) feedback networks. In a feedforward network, the connections between the neurons do not form a closed cycle and move in only one direction. A Hopfield network forms a feedback network, which is any network that is not a feedforward network. Furthermore a Hopfield network is fully connected, in which each neuron is connected to each other neuron. A key feature of Hopfield networks is that their dynamics tend to settle to a number of stable states, or "memories."

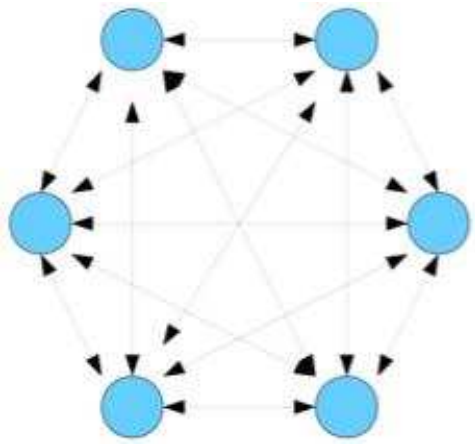

Fig. 2. A feedback network

For this project, we implemented a *binary* Hopfield network in which each neuron's activity is -1 or 1. This is a special case of a *continuous* Hopfield network where the neurons' activities are real numbers between -1 and 1.

1) Architecture: A Hopfield network is formed by I neurons which are fully connected and symmetric (see Figure 2). The weight from neuron i to neuron j is the same as from j to i (i.e. $w_{ij} = w_{ji}$). The output of the ith neuron will be denoted x_i.

2) Activity Rule: The activity of each neuron of a binary Hopfield network's is governed by the threshold function.

$$x(a) = \Theta(a) \equiv \begin{cases} 1 & a \geq 0 \\ -1 & a < 0 \end{cases} \tag{2}$$

Since this is a fully connected network, the activation of one neuron will affect the others. Thus we will need to update the network in one of two ways (*asynchronous* or *synchronous* update). With asynchronous update, each neuron will compute its activation and update in sequence. In synchronous update, all of the activations are computed and, only then, will the updates be calculated.

3) Learning Rule: In training a Hopfield network, our goal is create a set of desired stables states ($\mathbf{x}^{(n)}$). We can state it as follows: "for every pattern $\mathbf{x}^{(n)}$, if the neurons other than i are set correctly for $x_j = x_j^{(n)}$ then the activation of neuron i should be such that its preferred output is x_i."[1] The weights, $\mathbf{W}$, in a Hopfield network are determined via Hebbian learning. Hebbian learning was named after Donald Hebb, a Canadian psychologist who first postulated the idea in 1949. The essential idea behind Hebbian learning is that the weights between two positively correlated neurons should be increased. Thus activation of one neuron will likely lead to the activation of its associated neurons. Note that one neuron cannot be correlated with itself and thus will have a weight of zero. This is implemented mathematically using Hebb's rule.

$$w_{ij} = \eta \sum_n x_i^{(n)} x_j^{(n)} \tag{3}$$

$$w_{ii} = 0 \tag{4}$$

In this case, η is an unimportant constant. We'll set it to $1/N$, where N is the number of memories we are looking to store. Note that Hebb's rule will will create an IxI symmetric weight matrix.

We can increase the capacity of our network by modifying our objective function to measure how well it stores memories and minimizing that function. Consider the error between a single neuron of the network, with inputs $\{\mathbf{x^{(n)}}\}_{\mathbf{n=1}}^{\mathbf{N}}$ and binary labels $\{t^{(n)}\}_{n=1}^{N}$. We can define this error function $G(\mathbf{W})$ as follows [1]:

$$G(\mathbf{W}) = -\sum_n \left[t^{(n)} \ln y(^{(\mathbf{n})}; \mathbf{W}) + (1 - t^{(n)}) \ln(1 - y(^{(\mathbf{n})}; \mathbf{W})) \right] \tag{5}$$

where $y(\mathbf{x^{(n)}}_{n=1}; \mathbf{W})$ is the output of the neuron and

$$t_i^{(n)} = \begin{cases} 1 & x_i^{(n)} = 1 \\ 0 & x_i^{(n)} = -1. \end{cases} \tag{6}$$

Summing this over all I neurons we have the error for the entire network.

$$G(\mathbf{W}) = -\sum_i \sum_n \left[t^{(n)} \ln y(^{(\mathbf{n})}; \mathbf{W}) + (1 - t^{(n)}) \ln(1 - y(^{(\mathbf{n})}; \mathbf{W})) \right] \tag{7}$$

Now we can minimize $G(\mathbf{W})$ to determine our weights. However, we run the risk of overfitting our network. We will reduce this risk by making use of regularization. In this case we will minimize

$$M(\mathbf{W}) = G(\mathbf{W}) + \alpha E_W(\mathbf{W}) \tag{8}$$

where α is a regularization parameter and $E_W(\mathbf{W})$ is a penalty function [2]. This regularizer will act as a bias against $\mathbf{W}$ which will keep the values from becoming too large and overfitting the data.

We will be minimizing the function by utilizing gradient descent, a first order optimization algorithm. Gradient descent works via a backpropagation algorithm whereby we modify $\mathbf{w}$ to move opposite the gradient of G. In this way we find the $\mathbf{w}$ which minimizes the error function. The specific algorithm used is described in the appendix.

4) Convergence: To demonstrate convergence of a Hopfield network, we need to borrow a page from statistical physics. If we imagine the neural network to be a spin system in which each node could have two possible spins (± 1). It can be derived that the variational free energy of the system is approximated by the function [1]

$$\beta \tilde{F}(\mathbf{x}) = -\beta \frac{1}{2} \mathbf{x}^T \mathbf{W} \mathbf{x} - \sum_i H_2^{(e)}[(1 + x_i)/2] \tag{9}$$

where β is a constant from the probability distribution of initial spins and

$$H_2^e(q) = q \ln(\frac{1}{q}) + (1 - q) \ln[1/(1 - q)] \tag{10}$$

Function (9) is a *Lyapunov function*. This is important because if a dynamical system has a Lyapunov function, then its dynamics will settle to a fixed point (i.e. the local minimum). Therefore, due to the fact that a Hopfield network has such a function, it will eventually settle to its lowest energy state.

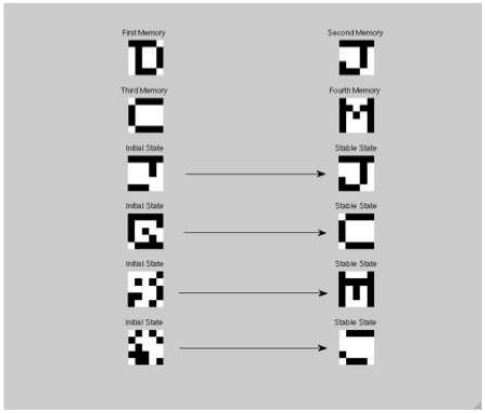

Fig. 3. Binary Hopfield Network Storing Four Memories

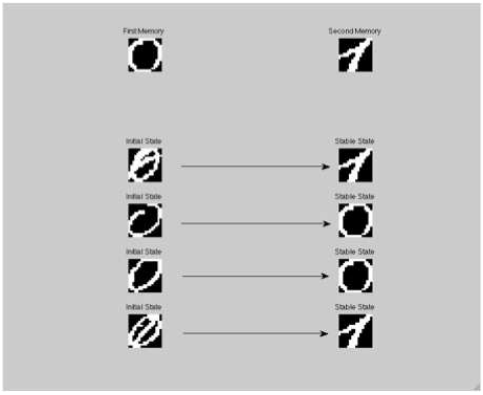

Fig. 4. Illustrative Convergence with Two Memories Stored

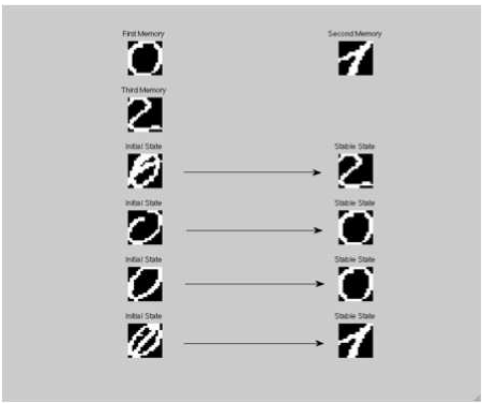

Fig. 5. Illustrative Convergence with Three Memories Stored

III. Results and Conclusions

A. Toy Problem

As a first attempt at coding a Hopfield network, we decided to code the Binary Hopfield network described in the book by MacKay [1]. Four memories are stored in this network, the letters D, J, C and M. Each of the memories are represented by a twenty-five element array and are shown as a five by five block of bits that are either white and "turned on", or black and "turned off". We then tested our code by introducing the same initial states with errors into our network as were introduced in the book. We found that most of our initial states did converge to the correct stable state. We also found that a couple of our initial states converged to a better stable state than the stable state presented in the text. However, we also found that some initial states converged to a mixed stable state, where none of the memories was explicitly present in the stable state, but a combination of the memories was produced. Figure 3 displays the four memories stored, four initial states, and the four stable states to which the initial states converged.

Looking at Figure 3, it can be seen that the first two initial states converged perfectly to a stored memory. It can also be seen that the third initial state converged to a stable state that differs by only two bits from one of the stored memories. The fourth initial state can be seen not converging to one of the stored memories, but rather a mixture of the stored memories.

After observing our results with storing four letter memories, we decided that our code was correct and that we could expand our code so that it would apply to the digit data set we wished to use.

B. Handwritten Data Set

Our data set is the Semeion Handwritten Digit Data Set which was obtained from the UCI Machine Learning Repository. The data set consists of 1,593 writing samples taken from 80 people. Each participant was told to write two copies of each of the digits zero through nine. One of the copies was supposed to be written as "carefully" or correctly as possible, while the other copy was to be written as "fast" as possible, or with errors. Each written digit was then fit into a 16x16 box, and was represented as a 256 element array where each element is represented by binary data [3].

After coding our algorithm, we decided to visually inspect what a few of our results looked like. Consider the convergence of four different samples of the digit zero. We started with storing two memories (digits 0 and 1), and then looked at the convergence of the same four sample digits with storing three memories (digits 0, 1 and 2), four memories (digits 0, 1, 2 and 3), and finally ten memories (digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9).

In Figure 4 it can be seen that the second and third sample zeros converge to the correct stored memory, the stored zero. However, the first and fourth sample zeros converge to the wrong stored memory, the one. We believe this occurs because the first and fourth sample zeros share more characteristics, such as the diagonal dash, with the one stored memory. In addition, it is possible that one stored memory could be a lower energy state and thus an easier convergence.

In Figure 5 it can be seen again that the second and third sample zeros converge to the correct stored memory, but that the first and fourth sample zeros converge to the wrong stored memory. However, in this case, the first sample zero converges to the two stored memory rather than the one stored memory. This again could be because it shares more characteristics with the two stored memory rather than the zero stored memory, or that the two stored memory is a lower energy state. It can also be seen that the forth sample zero still converges to the one stored memory.

In Figure 6 it can be seen that the second, third and fourth

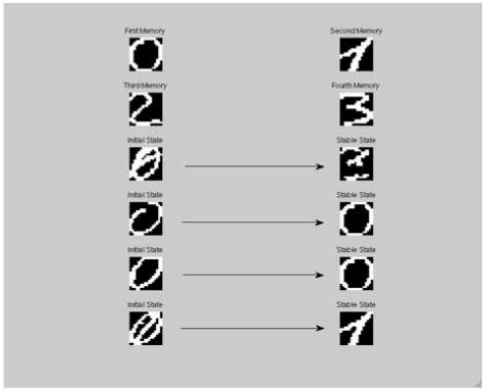

Fig. 6. Illustrative Convergence with Four Memories Stored

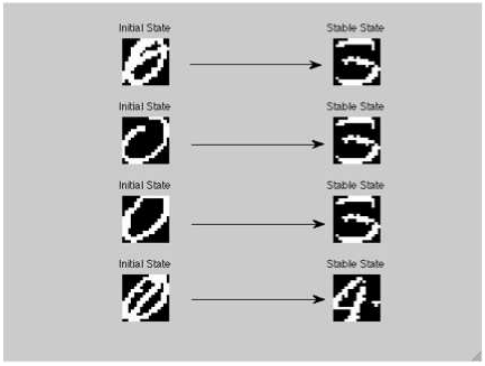

Fig. 7. Illustrative Convergence with Ten Memories Stored

sample zeros converge to the same stored memories as before in figures 4 and 5. However, in this case, the first sample zero converges to some kind of mixed stable state. The stable state is not identical to one of the stored memories, but has the same characteristics from more than one of the stored memories. This could again be due to the stable state being a lower energy state.

In Figure 7 it can be seen that none of the sample zeros converge to one of the stored memories. The first three converge to the same mixed stable state, and the fourth sample converges to a stable state that closely resembles a four, but is not exact and is thus a different mixed stable state. We found the fact that the first three samples converge to the same mixed stable state interesting, and believe that that particular mixed stable state must be dominant when running our algorithm with this data set. It is most likely one of the lowest energy states in the entire network.

C. Summary Results

After looking at our illustrative results, we saw an interesting trend. We observed that as the number of memories stored increases, the number of samples that converge to the correct stored memory decreases. This trend was expected due to the variation of our data set, and the nature of Hopfield networks in general. To better illustrate this trend, we decided to run through our data and record the percentage of correct convergences when storing two, three, five, nine and ten digits. We also decided to do this using the standard weights method,

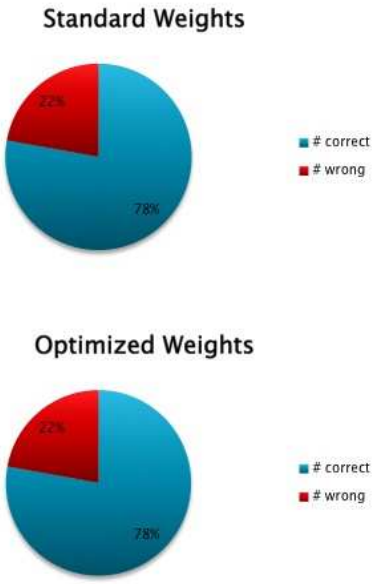

Fig. 8. Summary Results with Two Memories Stored

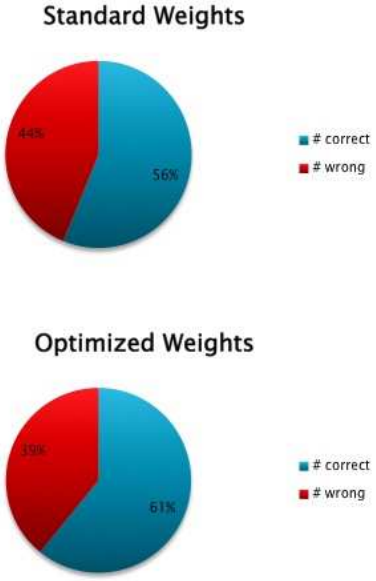

Fig. 9. Summary Results with Three Memories Stored

and the optimized weights method, in hopes of seeing any kind of improvement when using the optimized weights.

Looking at the first two pie charts with only two memories stored in Figure 8, it can be seen that the percentage correct using the standard weights and optimized weights is the same at 78%.

In Figure 9, when storing only three memories, that percent correct for the standardized weights drops to 56%, and the percent correct for the optimized weights jumps to 61%. Thus both percentages have dropped, but it can be seen that the optimized weights is performing slightly better.

Looking at Figure 10, with five memories stored, the percent correct for the standard weights drops to 49% and for the

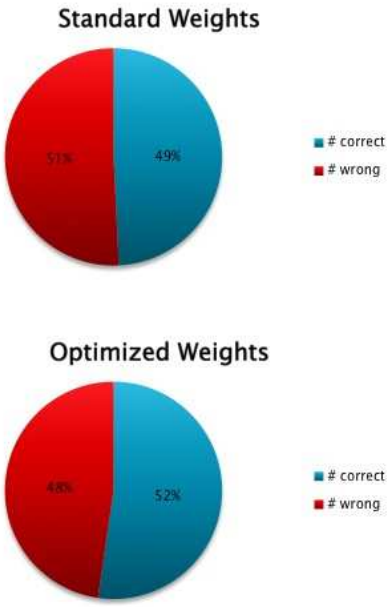

Fig. 10. Summary Results with Five Memories Stored

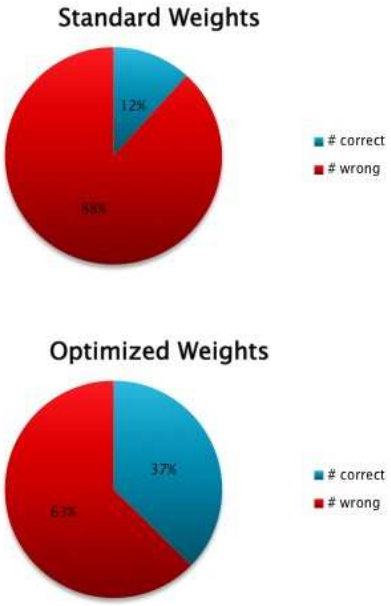

Fig. 11. Summary Results with Nine Memories Stored

optimized weights drop to 52%. This is a smaller gap between the performance of the standard vs optimized weights, but the optimized weights are still outdoing the standard weights.

In Figure 11, with nine memories stored, the percent correct plummets to 12% for the standard weights and 36% for the optimized weights. This is a much more noticeable gap between the performance of the standard and optimized weights.

Similarly, in Figure 12, with ten memories stored, the percent correct drops to 10% for the standard weights and remains at 36% for the optimized weights. This again shows that the optimized weights perform better than the standard weights.

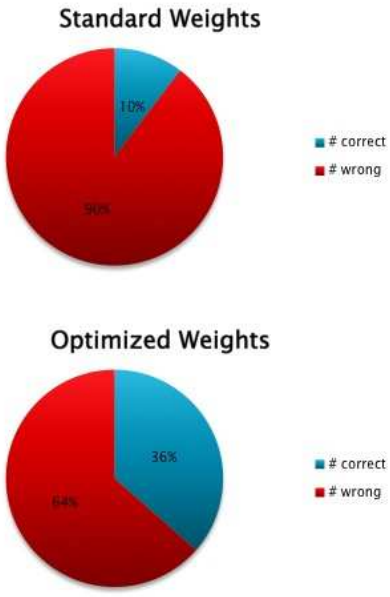

Fig. 12. Summary Results with Ten Memories Stored

IV. Conclusion

Hopfield networks are an easily implemented neural network that can effectively demonstrate associative memory. Partial or corrupted memories can be restored to their correct state. In addition, inputs that are sufficiently close to a stored memory will converge to the desired memory. The choice of weights is key in ensuring the highest rates of accuracy. Hebbian learning optimized via gradient descent seems to be the best choice to optimize the weights while preventing overfitting of the neural network.

Acknowledgment

The authors would like to thank Professor Ashish Bhan for teaching us all we know about data mining.

References

[1] David J. C. MacKay, *Information Theory, Inference, and Learning Algorithms*. New York, NY: Cambridge University Press, 2003.

[2] Alan Julian Izenman,*Modern Multivariate Statistical Techniques: Regression, Classification, and Manifold Learning*. New York, NY: Springer, 2008.

[3] Massimo Buscema, *Semeion Handwritten Digit Data Set*. Irvine, CA: UCI Machine Learning Repository [http://www.ics.uci.edu/ mlearn/MLRepository.html], 2003

APPENDIX A
MATLAB IMPLEMENTATION OF TOY PROBLEM

A. Toy Problem

```
function letterex()

%load data
D = [0 0 0 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1 1 0 1 0 0 0 1];
X = reshape(D, 5, 5);
subplot(6,2,1);
imshow(X');
title('First Memory');

J = [0 0 0 0 0 1 1 1 0 1 1 1 1 0 1 0 1 1 0 1 0 0 0 1 1];
Y = reshape(J, 5, 5);
subplot(6,2,2);
imshow(Y');
title('Second Memory');

C = [1 0 0 0 0 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 0 0 0 0];
Z = reshape(C, 5, 5);
subplot(6,2,3);
imshow(Z');
title('Third Memory');

M = [0 1 1 1 0 0 0 1 0 0 0 1 0 1 0 0 1 1 1 0 0 1 1 1 0];
P = reshape(M, 5, 5);
subplot(6,2,4);
imshow(P');
title('Fourth Memory');

%create data matrix
data(1,:) = D;
data(2,:) = J;
data(3,:) = C;
data(4,:) = M;
n = 25;        %number of data points in each picture

%make all 0's in data into -1's
datamones = data - ones(size(data));
newdata = data + datamones;
newdata
size(newdata)
%make weights matrix (n x n)
w = zeros(n,n);

for i = 1:n
    for j = 1:n
        if i == j
            w(i,j) = 0;
        else
            w(i,j) = newdata(:,i)'*newdata(:,j);

        end
    end
end
%initial states (with errors)
Xerr(5,:) = [-1 -1 -1 -1 -1 1 1 1 -1 1 1 1 1 -1 1 1 1 1 1 1 -1 -1 -1 1 1];
Xerr(7,:) = [-1 -1 -1 -1 -1 -1 1 1 1 -1 -1 1 -1 1 1 -1 1 1 -1 1 1 -1 ...
    -1 -1 -1];
Xerr(9,:) = [1 1 1 1 -1 1 -1 1 -1 1 1 1 1 1 -1 -1 -1 1 1 -1 -1 1 1 -1 1];
Xerr(11,:)= [1 -1 1 -1 1 -1 1 1 1 -1 1 -1 1 1 1 -1 -1 -1 1 1 1 -1 -1 1 -1];

for h = 5:2:11
Xerror = Xerr(h,:);
Xtest = Xerror;
X_new = Xtest;
X_old = ones(size(Xtest));
count = 0;
while X_old*X_new'~=length(Xtest)
    count = count +1
    % step 3: Asynchonous update
    if count == 50
        break
    else
    X_old = X_new; % previous new is now old
```

```
    X_new = AUpdate(X_old,w);
    end
end
%plot initial state and the stable state it converges to
X1 = reshape(Xerr(h,:), 5, 5);
subplot(6,2,h);
imshow(X1');
title('Initial State');

X = reshape(X_new, 5, 5);
subplot(6,2,h+1);
imshow(X');
title('Stable State');
end
end
```

APPENDIX B
MATLAB IMPLEMENTATION OF HANDWRITING CLASSIFIER

The following code forms the main subroutine in `Matlab`

```
% Main loop to store memories in a Hopfield Network and compare to
% handwriting samples

% load handwriting data
clear all
load semeion.data
DATA = semeion(:,1:256);                % 1st 256 columns contains image
index = numindex(semeion(:,257:266));   % create index of which number is
                                        % contained in each sample

% step 1 prepare data by converting all 0s to -1s, one additional column
% indicates number written
DP = [dataprep(DATA) index(:,2)];       % NxI+1

% Divide data into 10 matrices containing one number each
H_zeros = DP(index(:,2)==0,:);
H_ones = DP(index(:,2)==1,:);
H_twos = DP(index(:,2)==2,:);
H_threes = DP(index(:,2)==3,:);
H_fours = DP(index(:,2)==4,:);
H_fives = DP(index(:,2)==5,:);
H_sixes = DP(index(:,2)==6,:);
H_sevens = DP(index(:,2)==7,:);
H_eights = DP(index(:,2)==8,:);
H_nines = DP(index(:,2)==9,:);

% Matrix of stored memories
memories = [H_zeros(1,:); H_ones(1,:); H_twos(1,:); H_threes(1,:); ...
    H_fours(1,:); H_fives(1,:); H_sixes(1,:); H_sevens(1,:); ...
    H_eights(1,:); H_nines(1,:)];
[m n] = size(memories);
% Matrix of validation data
testdata = [H_zeros; H_ones; H_twos; H_threes; H_fours; H_fives; ...
    H_sixes; H_sevens; H_eights; H_nines];
[N n] = size(testdata);

% step 2 determine weights
% comment the learning rule you do not wish to use
%W = hopweights(memories(:,1:256));      % hebb's rule

% determine weights with modified formula for increased storage
W = hopweights2(memories(:,1:256));     % with gradient descent

% test loop
results = zeros(N,2);   % results of test first column is a 1/0
                        % indicating a correct answer
                        % the second column indicates the number of
                        % flipped bits

for i = 1:N             % N test samples
    Xtest = testdata(i,1:256);      % Test image
    testNumber = testdata(i,257);   % Test number written
    X_new = Xtest;
    X_old = ones(size(Xtest));
    count = 1;          % number of iterations in update loop
```

```
    % our stopping condition is where there is no change in states
    % between iteration.
    while X_old*X_new'~=length(Xtest)
        count = count + 1;
        % step 3: Asynchronous update
        if count == 100
            break
        else
            X_old = X_new; % the new image is now is now old
            X_new = AUpdate(X_old,W);   % perform asynchronous update
        end
    end
    % compare X_new to memories to determine difference
    bits = zeros(m,1);
    for j = 1:m
        diff = X_new.*memories(j,1:256);    % 1 or -1
        bits(j) = length(diff(diff~=1));    % # of flipped bits
    end
    % choose closest match to memory
    [bitmin match1] = min(bits);
    [bitmax match2] = max(bits);
    if memories(match1,257) == testNumber
        results(i,1) = 1;       % note that it's correct
        results(i,2) = bitmin;  % store minimum # of bits flipped
    elseif bitmax == 256        % in case inverse of memory is stable
        results(i,1) = 1;       % note that it's correct
        results(i,2) = bitmax;  % store number of bits off
    else                        % incorrect
        results(i,2) = 1000;
    end
end
```

The following function produces an index of all the handwritten numbers and where they occur in the Semeion data set.

```
function index = numindex(D)
%
% function index = numindex(D)
%
% This function calcutes the index of each from the Semeion
% handwriting data
%
% INPUTS:
%       D:      nX10, binary vector
% OUTPUTS:
%       index2: nX2, first value is the index of the number
%                    second is the number in image
[m n] = size(D);
index = zeros(m,2);     % initialize index matrix

for j = 1:m             % m is number of data points
    if D(j,:) == [1 0 0 0 0 0 0 0 0 0]      % if 0
        index(j,:) = [j 0];
    elseif D(j,:) == [0 1 0 0 0 0 0 0 0 0]  % if 1
        index(j,:) = [j 1];
    elseif D(j,:) == [0 0 1 0 0 0 0 0 0 0]  % if 2
        index(j,:) = [j 2];
    elseif D(j,:) == [0 0 0 1 0 0 0 0 0 0]  % if 3
        index(j,:) = [j 3];
    elseif D(j,:) == [0 0 0 0 1 0 0 0 0 0]  % if 4
        index(j,:) = [j 4];
    elseif D(j,:) == [0 0 0 0 0 1 0 0 0 0]  % if 5
        index(j,:) = [j 5];
    elseif D(j,:) == [0 0 0 0 0 0 1 0 0 0]  % if 6
        index(j,:) = [j 6];
    elseif D(j,:) == [0 0 0 0 0 0 0 1 0 0]  % if 7
        index(j,:) = [j 7];
    elseif D(j,:) == [0 0 0 0 0 0 0 0 1 0]  % if 8
        index(j,:) = [j 8];
    else
        index(j,:) = [j 9];                 % if 9
    end
end
```

The following function prepares the data to be used in a Hopfield network. The Semeion handwriting data is a 0/1 binary data. However, we require our data to be ± 1.

```
function DH = dataprep(DATA)
%
% function DH = dataprep(DATA)
%
% this function converts binary data (0,1) into the proper format for a
% Hopfield network (-1,1)
%
% INPUTS:
%       DATA:   256xN, binary data of handwritten images
% OUTPUTS:
%       DH:     256xN, binary data converted to 1 & -1

D_mod = DATA - ones(size(DATA));    % modified to include only 0 and -1
DH = DATA + D_mod;                  % 1s are still 1s and 0s are -1s
```

The following function computes the weights of a Hopfield network via Hebb's rule.

```
function W = hopweights(DATA)
%
% function W = hopweights(DATA)
%
% This function determines the weights of a Hopfield network vis Hebbian
% learning.
%
% INPUTS:
%       DATA:   256xN, binary data of handwritten images
% OUTPUTS:
%       W:      256x1, neurons' weights
[N dim] = size(DATA);
eta = 1/N;                  % we chose to be 1/N
W = zeros(dim,dim);
for i = 1:dim
    for j = 1:dim
        if i == j
            W(i,j) = 0;     % neuron cannot be correlated with itself
        else
            Xi = DATA(:,i);
            Xj = DATA(:,j);
            W(i,j) = Xi'*Xj;
        end
    end
end
```

The following function computes the weights of a Hopfield network via a gradient descent optimization of Hebbian learning.

```
function W = hopweights2(DATA)
%
% function W = hopweights2(DATA)
%
% Hopfield weights via an optimized Hebbian learning process. Adapted from
% MacKay P. 516.
%
% INPUTS:
%       DATA:   NxI, N memories to store, I neurons (binary)
% OUTPUTS:
%       W:      NxN, optimized weight matrix
[N I] = size(DATA);     % I is the number of Nuerons, N is number of
                        % stored memories
X = DATA;               % initialize data
t = DATA;
% turns -1s to 0s (the reverse of dataprep.m)
for i = 1:N
    for j = 1:I
        if DATA(i,j) == -1
            t(i,j) = 0;
        end
    end
end
eta = 0.1;              % learning rate
alpha = 1;              % weight decay rate
% eta=0.1 and alpha=1 seem to be the best combination
W = X'*X;               % initialize weights via Hebb's rule
SoS_old = 0;
SoS_new =1;
%for i = 1:L
Wcount =0;
while abs(SoS_new-SoS_old)>0.01
    if Wcount == 10000
        break
    end
    SoS_new - SoS_old   % for monitoring purposes
    W_old = W;
    SoS_old = SoS_new;
    Wcount = Wcount+1
    for i = 1:I
        W(i,i) = 0;
    end

    a = X*W;                    % activations
    Y = 1./(1+exp(-a));         % outputs via logistic function
    e = t-Y;                    % errors
    gw = X'*e;                  % compute gradients
    gw = gw + gw';              % symmetrize gradients
    W = W + eta*(gw-alpha*W);   % make step
    SoS_new = sum(sum((W-W_old).^2,1)); % check sum of squares error
end
```

The following function performs asynchronous update of a Hopfield network.

```
function Xnew = AUpdate(X,W)
%
% function Xnew = AUpdate(X,W)
%
% This function performs asynchronous update of a Hopfield Network
%
% INPUTS:
%       X:      256x1, current status of memory
%       W:      256x1, neurons' weights
% OUTPUTS:
%       Xnew:   256x1, updated neural network
[m n] = size(W);
a = zeros(m,1);
Xnew = zeros(size(X));
for i = 1:m
    a(i) = W(i,:)*X'; % Compute ith nueron's activation
    % determine X(i) via Threshold function
    if a(i)>=0
        Xnew(i) = 1;
    else
        Xnew(i) = -1;
    end
end
```

A. *Figures*

The following code is used to display the visual convergence of four sample initial states.

```
function proj()

load semeion.data.txt; %load data

X = semeion;
XX = X(:,1:256);
%possible training set
data(1,:) = XX(2,:);    %"good" zero
data(2,:) = XX(25,:);   %"good" one
data(3,:) = XX(44,:);   %"good" two
data(4,:) = XX(64,:);   %"good" three
data(5,:) = XX(87,:);   %"good" four
data(6,:) = XX(104,:);  %"good" five
data(7,:) = XX(124,:);  %"good" six
data(8,:) = XX(143,:);  %"good" seven
data(9,:) = XX(165,:);  %"good" eight
data(10,:) = XX(180,:); %"good" nine

[n I] = size(data);
datamones = data - ones(size(data));
newdata = data + datamones;
t = data;
eta = 0.01;
alpha = 0.01;
w = newdata'*newdata;

S_old = 0;
S_new = 1;
wcount = 0;

while abs(S_new - S_old) > 0.01
    if wcount == 10000
        break;
    end
    S_new - S_old
    w_old = w;
    S_old = S_new;
    wcount = wcount + 1;
    %construct weight matrix
    for i = 1:I
        w(i,i) = 0;
    end

    a = newdata*w;
    y = 1./(1 + exp(-a));
    e = t - y;
    gw = newdata'*e;
    gw = gw + gw';
    w = w +  eta* (gw - alpha*w);
    S_new = sum(sum((w-w_old).^2,1));

end

XXones = XX - ones(size(XX));
newXX = XX + XXones;

Xerr(5,:) = newXX(1,:);     % "bad" zeros (with errors)
Xerr(7,:) = newXX(3,:);
Xerr(9,:) = newXX(4,:);
Xerr(11,:) = newXX(5,:);
%Xerror = newXX(22,:);    % "bad" one
% Xerror = newXX(45,:);    % "bad" two
% Xerror = newXX(63,:);    % "bad" three
% Xerror = newXX(83,:);    % "bad" four
% Xerror = newXX(103,:);   % "bad" five
% Xerror = newXX(123,:);   % "bad" six
% Xerror = newXX(144,:);   % "bad" seven
% Xerror = newXX(166,:);   % "bad" eight
% Xerror = newXX(181,:);   % "bad" nine

for h = 5:2:11
Xerror = Xerr(h,:);
Xtest = Xerror;
X_new = Xtest;
```

```
X_old = ones(size(Xtest));
count = 0;
while X_old*X_new'~=length(Xtest)
    count = count +1;
    % step 3: (A)synchonous update
    if count == 50
        break;
    else
    X_old = X_new; % previous new is now old
    X_new = AUpdate(X_old,w);
    end
end
count
%plot initial state and stable state it converges to
Y = reshape(Xerr(h,:),16,16);
subplot(4,2,h-4);
imshow(Y');
title('Initial State');

X = reshape(X_new, 16, 16);
subplot(4,2,h-3);
imshow(X');
title('Stable State');
end
end
```