Impact of Updating Strategies in A Hopfield Network and Exploring Other Network Parameters

DATE: 10 APRIL, 2018
NAME: VIVEK KOODLI UDUPA
CU USER NAME: VKOODLI
ECE 8720 SECTION 001/002
TAKEHOME #1

Introduction

The Hopfield neural network is a simple artificial network which is able to store certain patterns. The entire pattern can be recovered if the network is presented with only partial information. The learning algorithm stores a given pattern in the network by adjusting the weights.
All the nodes in a Hopfield network are both inputs and outputs, and they are fully interconnected. That is, each node is an input to every other node in the network. There is no connection from a node back to itself. The weight matrix of a Hopfield network is a symmetric matrix with zero diagonal elements.
The learning algorithm of the Hopfield network is unsupervised. The algorithm is based on the principle of Hebbian learning which states that neurons that are active at the same time increase the weight between them.
The nodes can be updated Synchronously or Asynchronously. What kind of impact will different update strategies have on the Hopfeild network?

Goals

1. Select an appropriate dataset for the Hopfield network
2. Develop a Hopfield network with different update strategies
3. Observe the impact of different update strategies on the performance of the network developed

Outcomes/Deliverables

1. Matlab code for the Hopfield network developed
2. Perfomance analysis of different update strategies on the network developed.

References

[1]https://www.eriksmistad.no/hopfield-network/
[2]http://web.cs.ucla.edu/~rosen/161/notes/hopfield.html
[3]http://www.web-us.com/brain/neur_hopfield.html
[4] Artificial Neural Network by Robert J. Schalkoff

1. INTRODUCTION:

A Hopfield network is a simple recurrent neural network which can store binary patterns. The network is comprised of non-linear units which are connected to every other unit in the network except itself. All the nodes in the network act as both input and output. Then network can recover the entire stored pattern if presented with a partial pattern. The Hopfield weight matrix is a symmetric matrix with zero diagonal elements. The algorithm is based on the principle of Hebbian learning which states that neurons that are active at the same time increase the weight between them. The nodes of the network can be updated synchronously or asynchronously. Asynchronous update is a more realistic update method as neurons do not all fire at the same rate.

2. INITIAL WORK

2.1 Designing the Data Set

The Hopfield network stores binary {0,1} or bipolar {-1, +1} patterns and it can recover these stored patterns from partial patterns fed into the network. I decided upon using bipolar patterns to design the Hopfield network. Adobe Photoshop was used to design a 11px x 8px .png images. These images would serve as input patterns to my network. I designed few alphabets which are used as stable (patterns that are learned by the network) and few other distorted alphabets which are used as test states, which are used to see whether the network can recall the actual patterns from the distorted patterns. Figure 1.1 represents few Stable states and Figure 1.2 represents distorted patterns.

Figure 1.1 Stable States

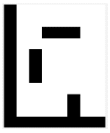

Figure 1.2 Test state

2.2 Importing the Patterns

Once the desired patterns were ready, they were imported into MATLAB and converted into a row vector of size 88 x 1. Then the binary values were converted into bipolar values for the ease of calculations.

2.3 Storing the pattern / Calculating the Weight Matrix

Synchronous method: The outer product of each pattern to be stored gives the weight matrix for that individual state. Sum of all such weight matrices gives the weight matrices which comprises of all the stable states/stored pattern.

Random Weight Update: Here a threshold is set, i.e 0.15 and uniform random numbers are generated for every weight in the network. Those weights whose random numbers are less than the threshold are updated using the Hebbian learning algorithm.

Note: It was observed that the weight matrix generated by both the methods yield the same result.(epoch = 200)

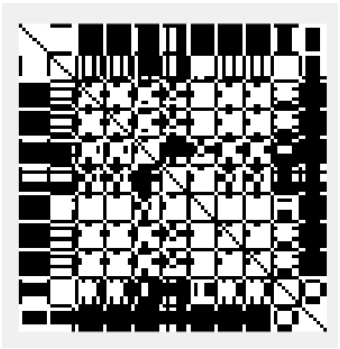

Figure 2.1 Weights from synchronous method

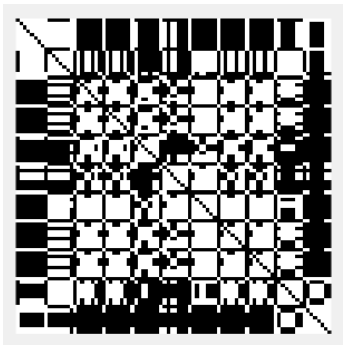

Figure 2.2 Weights from Random weight update

2.4 Checking the stored states

Now that the weight matrix was calculated, it is time to verify if the patterns are stored properly or not. Inner product of the stored state with the weight matrix will give the next state of the network. The inner product of the obtained state with the weight matrix gives the preceding state. This process is continued until the states stop changing its values. now if we check the values, they must correspond to one of the stored patterns. Figure 2.3 shows the recovery of stored pattern with the associated energy.

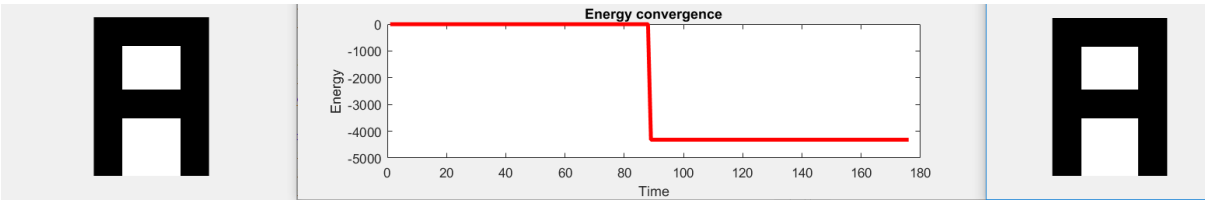

Figure 2.3 Testing the stored pattern 'A'(pattern on the left is the test pattern and the one on the right is the resultant pattern)

3. CHECKING THE CONVERGENCE OF DISTORTED PATTERNS.

After successfully verifying that the desired patterns were stored, the next step is to check if a distorted pattern can be converged into one of the stored patterns. I have implemented three methods to achieve this.

1. **Synchronous:** In this method, the next pattern is obtained by multiplying the entire test pattern vector with the weight matrix. i.e. All neurons are updated simultaneously. This process is repeated for epoch number of times or till results converge.
2. **Sequential:** Here, one node is updated at a time and with its updated value the next node is updated. An entire epoch is done when all the nodes are updated once. This process is repeated epoch number of times or till the results converge.
3. **Random:** In this process, a random sequence is generated and the nodes are updated as dictated by the generated sequence. This process is more realistic compared to the above two.

Results of the above methods are shown in Figure 3.1 through Figure 3.3

4. EXPLORING OTHER NETWORK PARAMETERS

1. **Unlearning:** Sometimes the network converges to spurious states. These are undesired states caused by the merging of Energy minima of very closely situated stable states. Such spurious states can be unlearnt by calculating the weight matrix for storing the spurious state and subtracting it from the main Weight matrix. The results are shown in Figure 4.1.x
2. **Over Crowding:** The network has a limit to the number of patterns it can store . Hopfield suggests that a d-neuron network allows approximately 0.15d stable states. My network is comprised of 88 neurons, and (0.15*88) is approximately 13. Ideally the network must store 13 patterns. But practically, it can store 5 patterns after which it fails to converge to desired states. Results are shown in Figure 4.2.x
3. **Deepening the energy well**: The energy of a certain stored pattern can be deepened by storing the same pattern multiple times. By doing so, the chances of convergence into such a state increases. Results are showin im figure 4.3.x

5. RESULTS

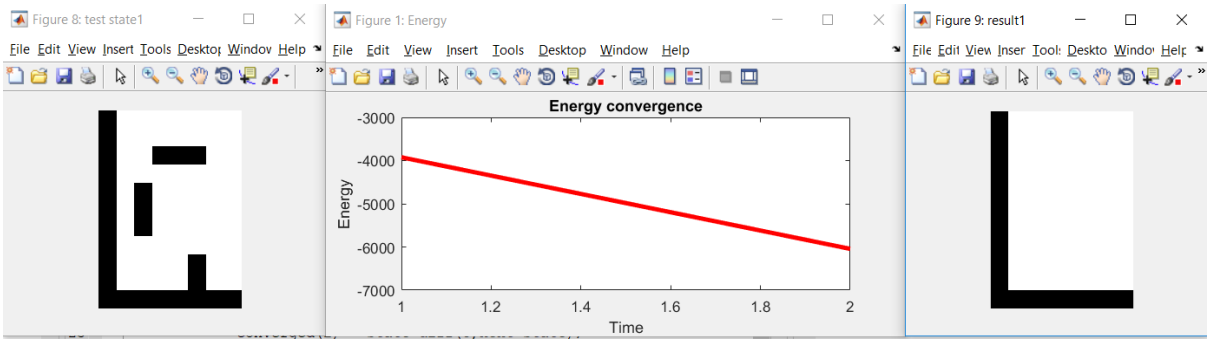

Figure 3.1.1 Distorted pattern converges to L(Stored Pattern) using the synchronous update method

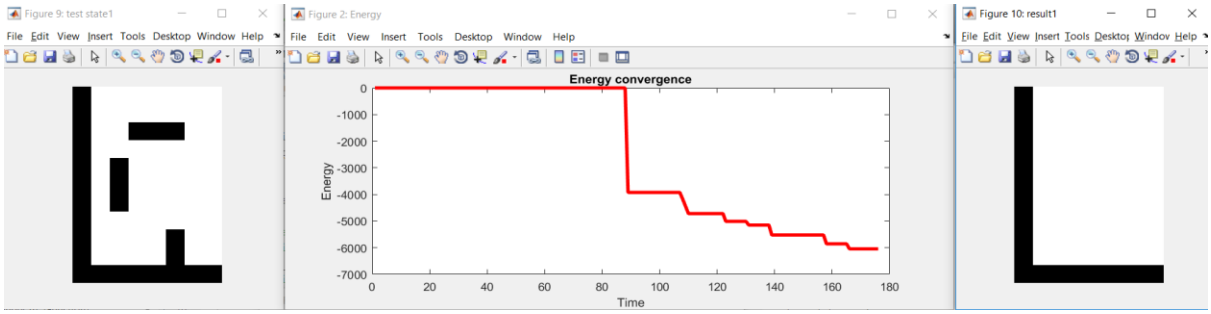

Figure 3.2.1 Distorted pattern converges to L using the sequential update method

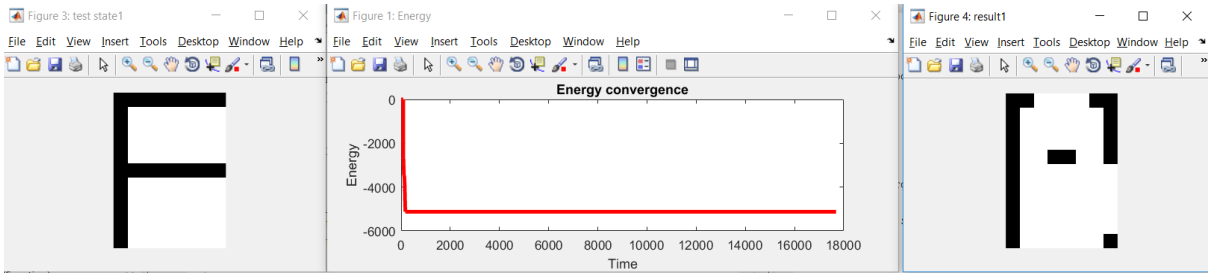

Figure 3.3.1 Test pattern ‘F’ Converging to Spurious state while using RANDOM update method

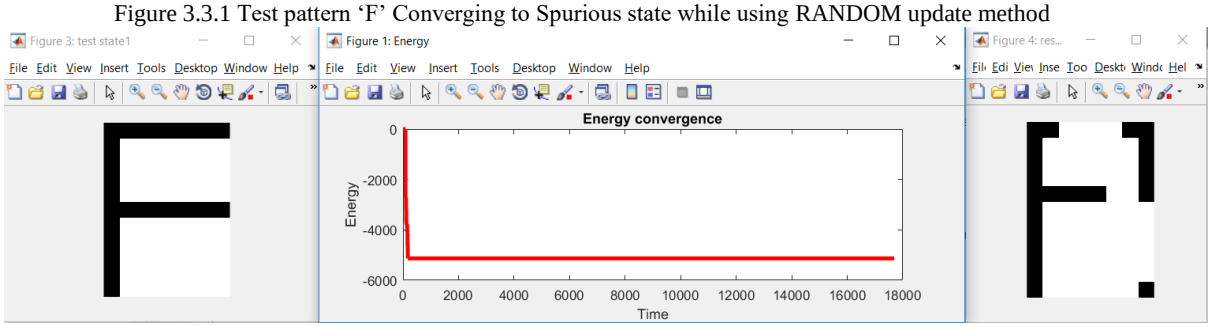

Figure 3.3.2 Convergence into slightly differnt pattern due to the change in the order of neuron update while using Random method

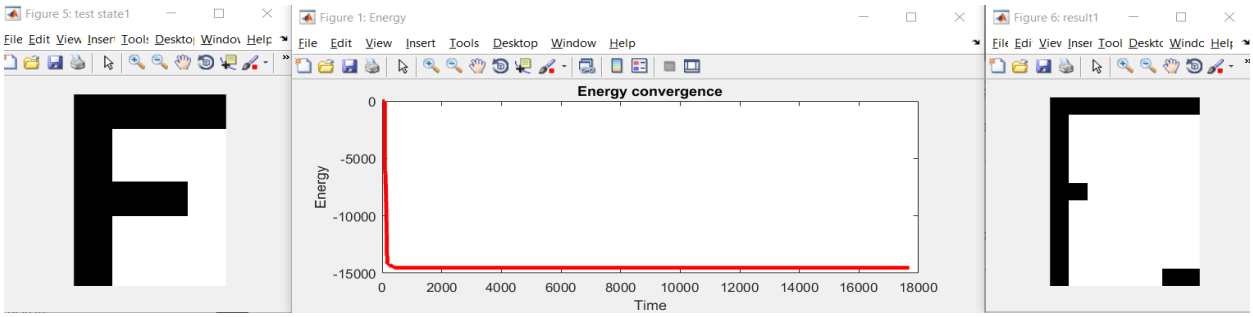

Figure 4.1.1 F Converging into a spurious state. (Note: F was not a stored pattern in this example)

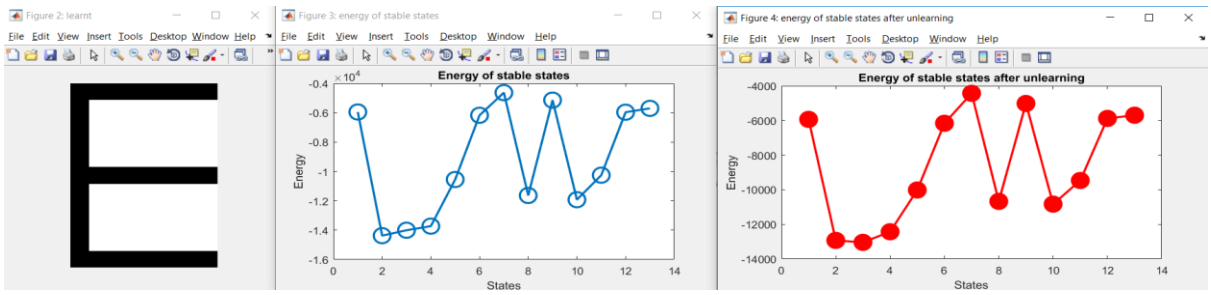

Figure 4.1.2 system converged to one of the stored pattern after unlearning.(left) The figure aslo shows the energies of stable states before(blue) and after(red) unlearning in 1D form.

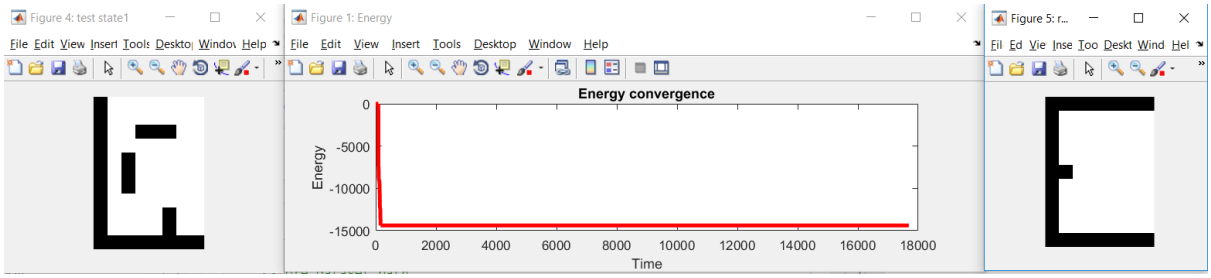

Figure 4.2.1 The same example used in Figure 3.2.1 failes to converge when number of stored states were increased from 5 to 13. Overcrowding the network's stable states results in spurious states.

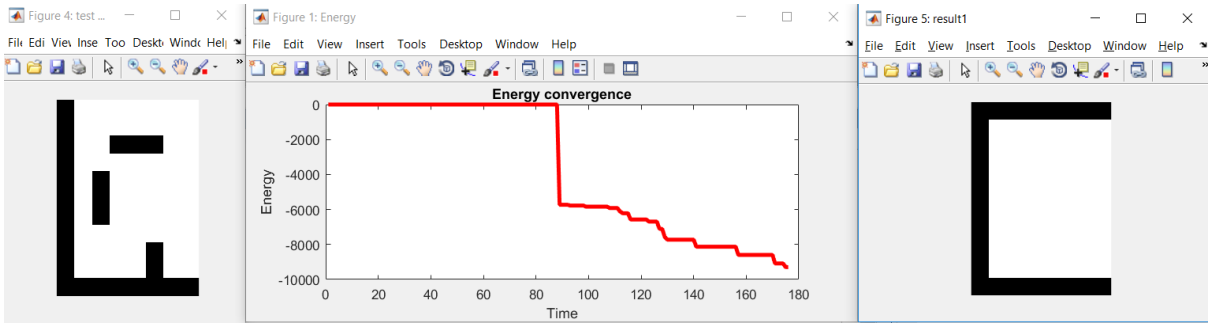

Figure 4.3.1: Convergence before deepening the energy well.

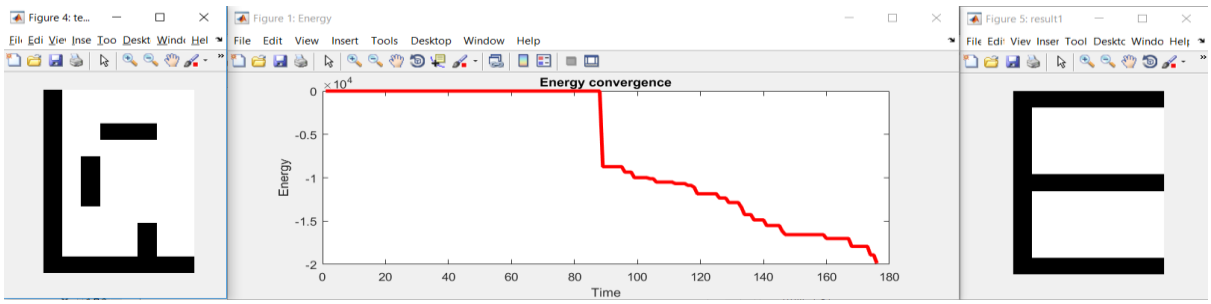

Figure 4.3.2 Convergene after stable state 3(i.e. 'E') has been stored 3 times

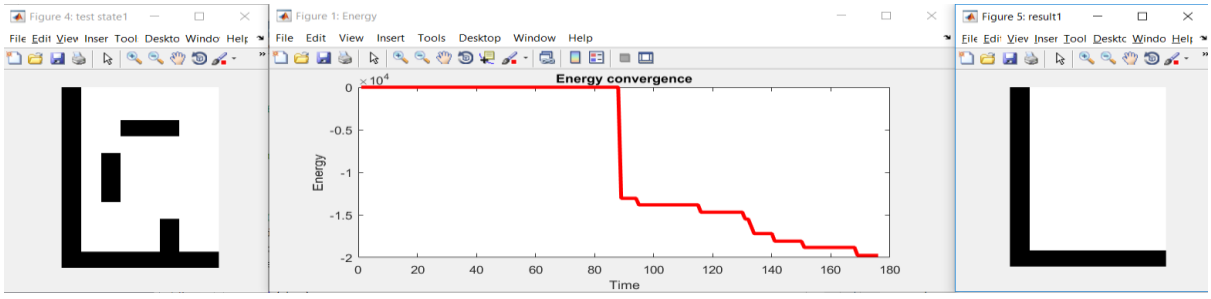

Figure 4.3.3 Convergene after stable state 5(i.e. 'L') has been stored 3 times

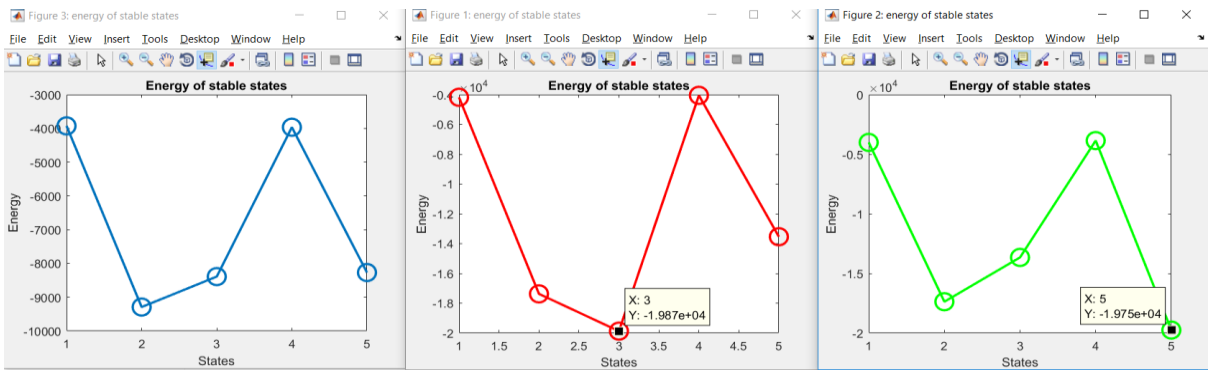

Figure 4.3.4 1D representation of energies of all stable states. Energies without deepening(BLUE), Energies with state 3 deepened(RED), Energies with state 5 deepened(GREEN)

6. CONCLUSION

Two neuron update strategies were implemented. Synchronous (synchronous method) and Asynchronous (sequential and random method). The synchronous method updates all its neurons simultaneously and generally converges into a stable state. The Asynchronous method, however, is more realistic. The neurons do not all fire at once. Each neuron is updated and its updated result is used in the next iteration. Depending on the order of neuron update, the convergence may vary. This can be observed in Figure 3.3.1 and Figure 3.3.2. A gif has been included in the submitted folder which shows the evolution of test state into a stable state.

Three network parameters were explored.

1. Unlearning: Network successfully unlearnt spurious state and converged into a stable state. This can be observed in Figure 4.1.1 and 4.1.2. Note that the test state 'F' used here was not a stable state. The network initially converged into a spurious state. But after learning the network converged into a stable state 'E'.
2. Over-crowding: The network designed can store 5 states and when overcrowded, it fails to converge into stable states, but rather converges into spurious states caused by the merging of energy minima's of close by stable states. 13 states were stored in this example. Results can be observed in Figure 4.2.1
3. Deepening the Energy well: Storing a stable state multiple times deepens the energy of that state. These deepened energy states tend to attract the test states towards themselves. Figure 4.3.1 shows the convergence of test state into a stable state normally. Observe how the same test state converges to a different stable state 'E' when the energy of 'E' is deepened. This is shown in Figure 4.3.2.

Figure 4.3.3 shows another such example by deepening the energy of stable state 'L'. Figure 4.3.4 represents the energies of all states in 1D form. It is observed that the energies of states 3 and 5 changes from approximately -9000 to -19000 when stored 3 times.

7. SUMMARY AND SUGGESTION FOR FUTURE WORK

A dataset of .png images of size 11px x 8px was created in Photoshop and was imported into MATLAB where the image was converted into a row vector of size 88 x 1. Later the data was converted from binary {0 1} format to bipolar {-1 1} format. The Weight matrix was calculated using the Hebbian learning rule. Neurons were updated with three different methods and the results were analyzed. Three network parameters namely Unlearning, Deepening the energy well and over-crowding was also implemented. Other network parameters such as diameter limited interconnections, biases, clipping the weight matrix etc can be implemented in the future.

APPENDIX

SYNCHRONOUS WEIGHT UPDATE CODE

```
1.    function W = Hopfield(o)
2.
3.       num_ss = length( o(1,:) ); %number os stable states
4.
5.       %storing individual states
6.       w = store(num_ss,o);
7.
8.       %Computing the Combined weight matrix
9.       W = combine(num_ss,w);
10.
11.      %Removing the diagonal elements
12.      W = rem_diag(W);
13.      end
14.
```

RANDOM WEIGHT UPDATE

```
function [W,E] = random(o,ep)


   %stable state is represented by the variable o

  num_units = length( o(:,1) );
  num_ss = length(o(1,:)); %number of stable states

  %initial random weights
  W = zeros(num_units,num_units);

  for p = 1: ep
     %selecting numbers < threshold
     nu_sel = rand(num_units,num_units);
     thres = 0.1;

     for x = 1:num_units %check all units
        for y = 1:num_units
           if(x == y)
              continue;
           elseif(nu_sel(x,y) > thres)
              continue;%skip if number is > threshold
           end
           %if number < threshold then go execute the following
           %% calculating the individual weight matrices for each stable state
           for i = 1:num_ss
%                w(i) = hebbian(num_ss,o,x,y,i);
              w(i) = o(x,i)*o(y,i);
           end %for i
           W(x,y) = sum(w);
           W(y,x) = W(x,y); %symmetric
           end %for y
     end %for x
     %calculate the energy
     for i = 1:num_ss
        E(i,p) = Energy(o(:,i),W);
     end
  end %for p/epoch
end %function
```

SYNCHRONOUS WEIGHT UPDATE

```
function [converged,test_final,E] = recover(W,test,o,epoch)

   x = length( test(1,:) ); %number of test entries
   converged = zeros(1,x);
   for i=1:x   %execute for each test data
      %converged vector holds value 1 in i'th column if i'th test
      %data converges to any of the stable state

      count = 1; %represents the number of executin taken to converge
      prev_state = test(:,i);
      while( count <= epoch ) %perform epoch number of times

         %Calculate the energy
         E(i,count) = -0.5 * prev_state' * W * prev_state;
%           figure(count+1000000)
%           imshow(vec_to_img(prev_state) );

         %calculate the next state
         next_state = W*prev_state;
         next_state = sign_corr(next_state,prev_state);

         %check if the next state converges to any stable state
         converged(i) = state_diff(o,next_state);


         count = count + 1;
         prev_state = next_state;

         %break the while loop if the test state converged to any stable
         %state
         if(converged(i) ~= 0)
            break;
         end


      end %while
      test_final(:,i) = prev_state;
      E(i,count) = -0.5 * prev_state' * W * prev_state;
   end %for
end
```

SEQUENTIAL WEIGHT UPDATE

```
function [test_final,E3,converged] = sequential(W,ss,tst,ep)
%This function performs sequential update of test images to check if they
%converge into any stable states

   num_ss = length( ss(1,:) ); %number of stable states
   num_unit = length( ss(:,1)); %number of units

   num_tst = length( tst(1,:) ); %number of test states
   converged = zeros(1,num_tst); %initialization

   for t = 1:num_tst %perform for all test states
      prev_state = tst(:,t);
      next_state = tst(:,t);
      for i = 1:ep %execute epoch number of times

         %show image evolution at each itiration
%           figure('name','evolution_sequential')
%           imshow(vec_to_img(next_state),'InitialMagnification','fit');

         for o = 1:num_unit
```

```
            next_state(o,1) = W(o,:) * prev_state ;
            next_state(o,1) = sign_corr( next_state(o,1), prev_state(o,1));
            prev_state = next_state;

            %calculate energy at each step
            E3(t,(num_unit*i + o)) = Energy(next_state,W);

        end %o

        %check for convergence into stable state
        converged(t) = state_diff(ss,next_state);
        %if the test state reaches a stable state then stop
        if(converged(t) ~= 0)
           break;
        end



     end% epoch
     test_final(:,t) = next_state;
  end %for t

end %function
```

RANDOM WEIGHT UPDATE

```
function [test_final,E2,converged] = asynchronous(W,ss,tst,ep)

   num_ss = length( ss(1,:) ); %number of stable states
   num_unit = length( ss(:,1)); %number of units

   num_tst = length( tst(1,:) ); %number of test states
   converged = zeros(1,num_tst); %initialization
   for t = 1:num_tst %perform for all test states
      prev_state = tst(:,t);
      next_state = tst(:,t);
      %generate a random order in which the network will update units
%       order = randperm( num_unit );

      for i = 1:ep %execute epoch number of times
         %generate a random order in which the network will update units
         order = randperm( num_unit );
%          %show image evolution at each itiration
%          figure('name','evolution_async')
%          imshow(vec_to_img(next_state),'InitialMagnification','fit');
%          a = strcat('evol_async',num2str(i));
%          saveas(gcf,a);

         for o = 1:num_unit
             %display fig
             %show image evolution at each itiration
%                figure('name','evolution_async')
%                imshow(vec_to_img(next_state),'InitialMagnification','fit');
%                a = strcat('evol_async',num2str((num_unit*i + o)));
%                saveas(gcf,a,'jpeg');
%                close;
             %end

             unt = order(o); %selct the unit to update from order
             next_state(unt,1) = W(unt,:) * prev_state ;
             next_state(unt,1) = sign_corr( next_state(unt,1), prev_state(unt,1));
             prev_state = next_state;

             %calculate energy at each step
             E2(t,(num_unit*i + o)) = Energy(next_state,W);
```

```
        end %o

        %check for convergence into stable state
        converged(t) = state_diff(ss,next_state);
        %if the test state reaches a stable state then stop
        if(converged(t) ~= 0)
           break;
        end



      end% epoch
      test_final(:,t) = next_state;
   end %for t

end %function
```

UNLEARNING

```
function W_new = unlearn(W,sp_state)
%This function will change the weight matrix by subtracting the weight
%matrix of the spurious state.

   a = 0.5 % learning rate
    W_new = W - a*(sp_state' .* sp_state); %subtracting the spurious state weights
%

end %function
```

---------- **END** ----------