Storage Capacity of Hopfield Memory Models

Alena Klindziuk

CAAM 588 Final Project

Abstract

This project investigates the difference in the storage capacity of two types of associative memory networks: a regular Hopfield network employing Hebbian learning and Hopfield network with Perceptron learning rule. The models are contrasted in terms of their storage capacity and associativity. To test the capacity, the relationship between the output error rate and the number of stored memories is investigated in both networks. For associativity, the influence of input mismatch rate on the output error rate is studied. Simulation experiments on a regular Hopfield network suggests a capacity of about 14 % of the input dimension, and good performance on associative tasks with small decreases in capacity for 10 % and 20 % mismatched inputs. Simulating the Perceptron-type network suggests a capacity that is 90 % of N, with poor performance on associative tasks.

1 Introduction

Associative memory is when a piece of information can be recalled by being prompted with a incomplete or noisy key, which chan be thought of as input. For example, being able to recall a whole phone number after being given the first few digits in the sequence requires associative memory. One of the models for associative memory is the binary Hopfield network [1]. A Hopfield network is a recurrent network where all the nodes are connected with one another and the inputs are guaranteed to converge to one of the attractors, which are the stored memories. Such a network may store P patterns of dimension N. The memorized patterns are stored in a square matrix W with elements w_{ij}.

The Hopfield can be implemented with various types of learning rules. For example, the original implementation uses Hebbian learning algorithm to obtain the weight matrix associated with a set of 'memorized' patterns. However, one of the drawbacks of Hopfiled network under Hebbian learning rule, is that it has a low storage capacity. the capacity of such a network is given by $p_{max} = 0.138N$, meaning that approximately 14 vectors can be recalled from storage in a network of 100 nodes. If P is at $0.138N$, about 0.37 % of the bits will flip initially, and it may be that the these flips will cause more bits to flip over subsequent iterations in recall [1]. In fact if $p_{max} \geq 0.138N$, the initial bit flips will cause an avalanche phenomenon where the recalled pattern bears no resemblance to the original pattern. If p_{max} = $0.138N$, 0.37 % of the bits will flip initially, but 1.6 % will flip before a stable attractor is reached [1]. There exists an alternative definition of capacity, which rests on the assumption that most of the memories are recalled perfectly. That is, the network recalls N bits right with 99 % probability. In such case, the the capacity is $p_{max} = N/\log N$ [4]. The capacity, p_{max} is proportional to N but is never higher than $0.138N$ is small percentage of error is acceptable in the recalled pattern, but is proportional to $N/\log N$ if all patterns need to be recalled perfectly. To increase the capacity of the network, other learning rules can be applied.

Using the Perceptron learning to train a Hopfield network increases the capacity to $2N$. However, in a special case that the threshold vector of a network is zero, it was proved that C(n)=n under each of the spectral strategies [3]. The Perceptron-type learning rule can be thought of as a supervised extension of the Hebbian learning.

This project is an experimental investigation of whether the Perceptron-type learning rule is indeed superior to Hebb in terms of storage capacity. One of the questions this study answers is which network is better at association tasks.

2 Methods

Both the networks perform two functions: storage of the memory patterns and recall of the memory patterns. They learn a set of N-dimensional binary patterns x^1, x^2, ..., x^P. During storage, each network encodes the pattern vectors into a weight matrix $\mathbf{w}$, and because of the symmetry of of the network connections weights, the diagonal elements or the weight matrix are 0. The regular Hopfield network uses a Hebb-type learning rule to adjust the weights, while the Perceptron-type network uses the Perceptron algorithm to ineratively determine $\mathbf{w}$.

In the Hebb-type rule, the weight matrix is adjusted as follows:

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \eta(\mathbf{x}^\mu)^T\mathbf{x}^\mu$$

Here, η is the learning rate specifying how quickly the rule gets updated, μ is an index of a specific pattern, t is the time step. This is an online rule, meaning that the patterns are learned one by one, and the weights from each are added to the matrix.

The perception-type rule may be described by the following training procedure:

$$\mathbf{w}(t+1) = \mathbf{w}(t+1) + \eta[\mathbf{d}(n) - \mathbf{y}(n)](\mathbf{x}(n)^\mu)^T$$

where

$$\mathbf{y}(n) = \mathrm{sgn}(\mathrm{w}\mathbf{x}^\mu)$$

and $\mathbf{b}$(n) is a vector of bipolar desired output values. The function sgn(x) assigns in whether the weights are increasing or decreasing according to:

$$\mathrm{sgn}(x) = \begin{cases} +1 & \text{for } x < 0 \\ -1 & \text{for } x > 0 \end{cases}$$

Because this is a system with a threshold, here set to 0, it is non-linear. Hence, during each time iteration, the weights need to be updated according to all the stored patters in the system. That is unlike in the regular Hopfield learning, where the weights from each pattern are simply added together, the perceptron-type learning rule loops through all patters during each learning iteration. After each time step, the values on the diagonal of $\mathbf{w}$ need to be set to 0, so that the matrix can be used in the Hopfield recall algorithm.

The Hopfield recall process works for both the network with Hebb-type learning scheme and the Perceptron-type learning scheme. To recall the stored vector, the following procedure is iteratively followed:

$$\mathbf{y}^\mu = sgn(\mathbf{w}(\mathbf{x}^\mu)^T$$

and each element of resulting vector $\mathbf{y}$ is put through function

$$\mathrm{sgn}(y_i) = \begin{cases} +1 & \text{for } y_j < 0 \\ -1 & \text{for } y_i > 0 \end{cases}$$

This algorithm takes as input the 'key' to one of the stored memories - a vector that is exactly alike or similar to one of the stored vectors. The process return a vector from the memory that is most similar to the input vector.

The two types of networks were simulated using MATLAB to explore the relationship between number of stored vectors, P, vs. the accuracy of recall, quantified by the number of bits in the output vector that do not match the bits in the input vector. The strength of association in two networks will also be compared. The network that is more strongly associative is the one which is able to recall memories with the greatest accuracy even if the input is corrupted, that is, a certain percentage of bits in the input vector does not match the bits in the corresponding output vector.

3 Results

Figure 1 shows how the output error rate varies with the number of stored vectors, P, for a one hundred 100-dimensional regular Hopfield networks with P from 1 to 100. The case for 100 % accurate input is on the left, 10 % corrupt input is in the center, and 20 % corrupt input is on the right. On all of the

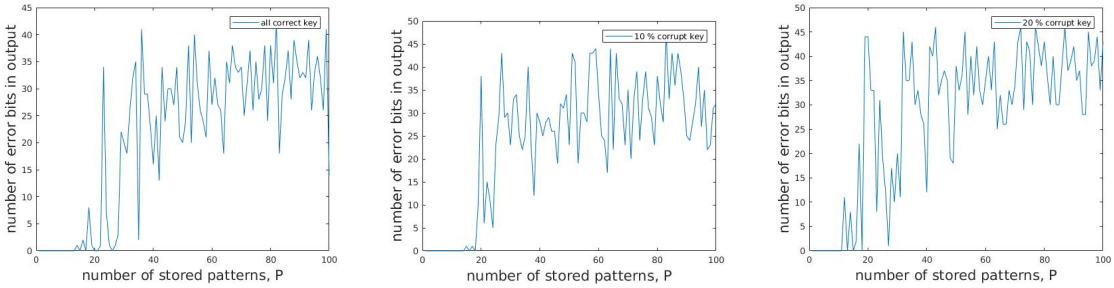

Figure 1: Simulation results for a Hopfield network. The stored patterns have dimension N = 100. Each data point (connected by a blue line) represents the number of error bits in the recalled pattern for various numbers of patters stored, P, from 1 to 100. Graph on the left shows a simulation where the input into the network is identical to one of the stored patterns. The case where 10 % of input bits are flipped is in the center, and the case for 20 % corrupted input input is on the right.

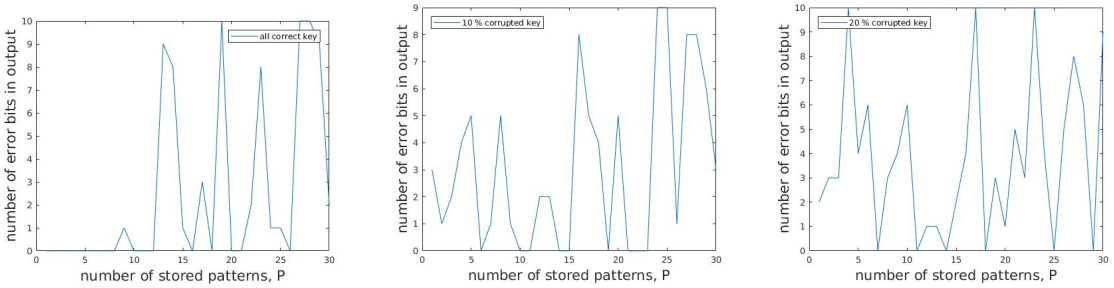

Figure 2: Simulation results for a Perceptron-type Hopfield network. The stored patterns have dimension N = 10. Each data point (connected by a blue line) represents the number of error bits in the recalled pattern for various numbers of patters stored, P, from 1 to 30. Graph on the left shows a simulation where the input into the network is identical to one of the stored patterns. The case where 10 % of input bits are flipped is in the center, and the case for 20 % corrupted input input is on the right.

graphs, the error rates rise rather sharply beyond a certain P, and level off at about an average of 32 % output bit flip ratio. In the left graph with all correct input, first output error occurs at $P = 14$. If 10 % of the input was mismatched initially, the first error occurs at $P = 13$ and with the 20 % corrupt input, $P = 12$ at a particular simulation instance . The finding in the first graph is is consistent with theory. According to theory, a regular $N = 100$-dimensional Hopfield network can only store about 14 patterns.

Figure 2 shows how the output error rate varies with P for thirty 10-dimensional Perceptron-type Hopfield networks with each network having a different value of P from 1 to 30. In the case for 100 % accurate input on the left, the fist bit flip in the output occurs at $P = 9$. And when the input has 10 % and 20 % of their bits flipped, the no correct output is returned. The theory predicts that the error in the output should occur at $P = 10$ for the network being considered. Hence, the simulation results do match the theory well. The one bit discrepancy can be attributed to the realatively small dimension of the network tested.

Figure 3 compares how well the two types of networks can perform association of memories. On the right, the Perceptron-type network fails to link the 20 % corrupted input to a learned pattern. Meanwhile, the regular Hopfield network is little affected by having 20 % of its inputs mismatch the stored patterns. The storage capacity drops from 15 % storage capacity to 12 % storage capacity in the given simulation. This demonstrates the fact that in spite of having a smaller storage capacity than a Perceptron-type Hopfield net, regular Hopfield nets are better at peforming associative tasks.

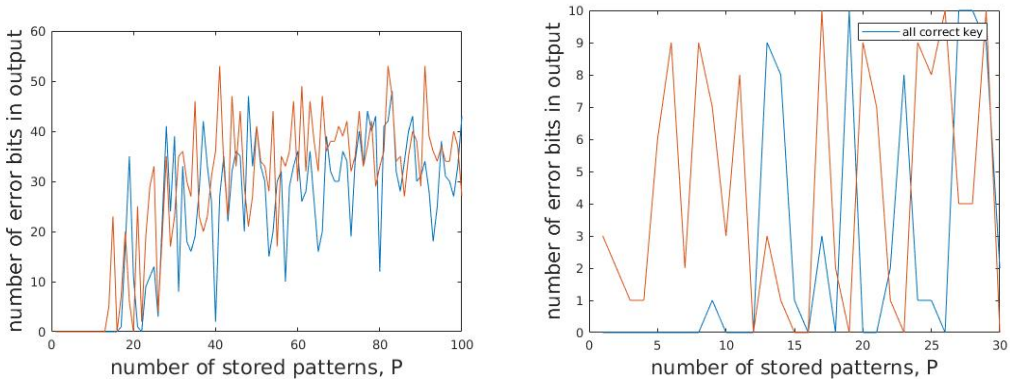

Figure 3: Comparison between the all correct input (blue) and 20 % corrupt input (orange) for the Perceptron-type Hopfield net (left) and the regular Hopfield net (left).

4 Conclusions

It can be concluded from the simulation results that the regular Hopfield have a memory capacity for correct recall of about 14 % of its dimension, as the thory suggests. However, the simulations extend these findings to show that even with 10 % and 20 % error rates in the input, the regular Hopfield net can match the input to the stored memory with little loss to the capacity. On the other hand, Perceptron-type Hopfield network, has a much higher capacity of about $0.9N$ according to the simulation. This is close to the theoretically predicted capacity of N for such a network. Since the implementation of Hopfield network used in this project has a threashold vector of 0, the simulation is consistent with the theory. So, even with a zero threshold, the Perceptron-type type network is superior to regular Hopfield net in terms of storage capacity. However, Hofield net with Hebbian learning can perform associative tasks better than the Percptron-type. It can not be concluded from the above results that the Perceptron-type Hopfield net can not perform associative tasks. It may be possible that such kind of network can be associative when only 5 % of its input has been corrupted. However, with 10-dimensional net, such scenario could not be tested, as one bit can not be split.

The Perceptron-type net was implemented with a relatively small dimension because at larger dimensions, the simulation would run out of memory. Hence one future direction for the work is to streamline the Perceptron-type Hopfield algorithm, and implement higher dimensional networks to see if Perceptron type net is associative. Also, It would be interesting to comare the computational efficiency of either type of network.

References

[1] J. Hertz, A. Krogh, R.G. Palmer, Introduction to the Theory of Neural Computation, Addison-Wesley, Reading, MA, 1991

[2] J.J. Hopfield. Neural networks and physical system with emergent collective computational abilities. Proceedings of the National Academy of Science, USA, 79 (1982), pp. 2554-2558

[3] S.S. Venkatesh, D. Psaltis. Linear and logarithmic capacities in associative memory. IEEE Transaction on Information Theory, 35 (1989), pp. 558-568

[4] R.E. McMliece, et al. The capacity of the Hopfield associative memory. IEEE Transactions on Information Theory, 33 (1987), pp. 461-483

5 Appendix

```
%simulation for the standard Hopfield netrwork

P = 100; %number of vectors
N = 100; %dimension of vectors
delta2 = zeros(P,1);

for m = 1:3
for k = 1:P % loop P, number of patterns
    n = N; %all vectors have same dimension
    p = k; %vary number of vectors
input = randi([0 1], p, n);%generate some vectors to memorize
q = size(input,1); % Find the number of vectors
n = size(input,2); % Find the dimension of the vectors
x = 2*(input) - 1;% Convert to bipolar

% Initialize and compute the weight matrix
w = zeros(n,n);
for i=1:q
 w = w + x(i,:)'*x(i,:);
end
w = w - q*eye(n); % Zero off the diagonal

%set the probe vector
Num = 1; %which vector to recall
probe = input(Num,:);

%flip a few bits
flip_N = 0;
for i =1:flip_N
    probe(i) = abs(probe(i)-1);
end

%ok now we got the signal
signal_vector = 2*probe-1; % Convert probe to bipolar form
flag = 0; % Initialize flag

while flag ~= n
    permindex = randperm(n); % Randomize the order of update
    old_signal_vector = signal_vector; % Keep a copy of the signals
    for j = 1:n  % Update all neurons once per epoch
      act_vec = signal_vector * w;
      if act_vec(permindex(j)) > 0
        signal_vector(permindex(j)) = 1;
```

```
      elseif act_vec(permindex(j)) < 0
        signal_vector(permindex(j)) = -1;
      end
    end
  flag = signal_vector*old_signal_vector'; % Generate flag
end
stored_original = input(Num,:); %whats in the memory
corrupted_input = probe; % with bits flipped - try to associate it
returned = .5*(signal_vector+1); % attempt to associate from incomplete probe
delta = stored_original - returned;
delta2(k,1) = delta*delta';
end
end

figure(1)
plot(delta2);
hold on
xlabel('number of stored patterns, P', 'FontSize', 16);
ylabel('number of error bits in output', 'FontSize', 16); %mismatch in the associated pattern to the pattern s

%simulation for perceptron-type Hopfield network

nth= 1; %which num vector to recall
N = 10; %set dimension
P = 30; %set num vec
eta = .1; %learning rate

for r = 1:P %iterate over total number of patterns stored
    P = r;

input = randi([0 1], P, N); %memorized pattern
p = size(input,1); % Find the number of vectors
n = size(input,2); % Find the dimension of the vectors
y = zeros(1,n);
w = rand(n,n);
x = 2*input-1; % Convert probe to bipolar form
d = x(nth,:); %desired out

for t = 1:100
  for k = 1:r
      Y = x(k,:)*w';
      y = sgn(Y);
      dw = eta*(d - y)'*x(k,:);
      w = w+ dw;
      for j = 1:n
          w(j,j) = 0;
      end
  end
end

%flip a few bits in the key (input)
%}
flip_N = 2;
    for i =1:flip_N
        d(i) = d(i)*(-1);
    end
%}

probe = d; %Read probe vector = desired out (ideally)
```

```
flag = 0; % Initialize flag ;
vector = probe;

while flag ~= n
    permindex = randperm(n); % Randomize the order of update
     old_signal_vector = vector;% Keep a copy of the signals

    for j = 1:n
      act_vec = w*probe';
      if act_vec(permindex(j)) > 0
        vector(permindex(j)) = 1;
      elseif act_vec(permindex(j)) < 0
        vector(permindex(j)) = -1;
      end
    end
  flag = vector*old_signal_vector'; % Generate flag
end

stored_original(r,:) = input(nth,:); %whats in the memory
returned(r,:) = .5*(vector+1); % attempt to associate from incomplete probe
delta1(r,:) = abs(stored_original(r,:) - returned(r,:));
delta2(r) = delta1(r,:)*delta1(r,:)'; %count number of mismatches
end

figure(2)
plot(delta2);
hold on
xlabel('number of stored patterns, P', 'FontSize', 16);
ylabel('number of error bits in output', 'FontSize', 16); %mismatch in the associated pattern to the pattern s
legend('all correct key');

function y = sgn(x)
    l = length(x);
    y = zeros(1,l);
    for i = 1:l
        if x(i) > 0
            y(i) = 1;
        elseif x(i) < 0
            y(i) = -1;
        end
    end
end
```