|  |  |
| --- | --- |
|  | CSC 424 Artificial Neural Network |
|  |  |
|  | Tanjina Piash Proma ID:1320810  Portfolio  8/6/16 |

Contents

[Topic 01: Hamming Network 1](#_Toc458414647)

[Matlab code for Hamming Network: 2](#_Toc458414648)

[Topic 02: Hopfield Network 3](#_Toc458414649)

[Matlab code for Hopfield Network: 3](#_Toc458414650)

[Topic 02: Supervised Hebbian Learning 4](#_Toc458414651)

[The Hebb Rule 4](#_Toc458414652)

[Pseudoinverse Rule: 4](#_Toc458414653)

[Auto Associative Memory: 4](#_Toc458414654)

[Accuracy of Hebb’s rule: 8](#_Toc458414655)

[Accuracy of Hebb’s rule: 8](#_Toc458414656)

[Matlab code for Auto Associative Network: 9](#_Toc458414657)

[Auto Associative Network using Hebbian rule: 9](#_Toc458414658)

[Auto Associative Network using Pseudoinverse rule: 10](#_Toc458414659)

[Topic 03: Backpropagation 11](#_Toc458414660)

[Matlab code for Backpropagation: 11](#_Toc458414661)

[Function Approximation: 12](#_Toc458414662)

[Topic 04: Competitive Network 13](#_Toc458414663)

[Matlab code for competitive network: 13](#_Toc458414664)

# Topic 01: Hamming Network

The objective of Hamming Network is to decide which prototype vector is closest to the input vector. The decision is indicated by the output of the recurrent layer.

****

## Matlab code for Hamming Network:

p1 = [1 -1 -1]';

p2 = [1 1 -1]';

p3 = [-1 -1 -1 ]';

%feed forward layer

w1 = [p1 p2]';

b1 = [3;3];

a1 = purelin(w1\*p3 + b1);

%recurrent layer

w2 = [1 -0.5;-0.5 1];

a2 = a1;

while true

new\_a2 = poslin(w2\*a2);

if new\_a2 ==a2

break;

else

a2 = new\_a2;

end

end

# Topic 02: Hopfield Network

The neurons in this network are initialized with the input vector, then the network iterates until the output converges. When the network operates correctly, the resulting output should be one of the prototype vectors.



Figure: Hopfield network

## Matlab code for Hopfield Network:

p1 = [1 -1 -1]';

p2 = [1 1 -1]';

p3 = [-1 -1 -1 ]';

w= [0.2 0 0; 0 1.2 0;0 0 0.2];

b = [0.9;0;-0.9];

a = p3;

a1= a;

while true

new\_a1 = satlins(w\*a1+b);

if a1== new\_a1

break;

else

a1 = new\_a1;

end

end

# Topic 02: Supervised Hebbian Learning

## The Hebb Rule

The Hebbian Learning Rule is a learning rule that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. The rule builds on Hebbs's 1949 learning rule which states that the connections between two neurons might be strengthened if the neurons fire simultaneously. The Hebbian Rule works well as long as all the input patterns are orthogonal or uncorrelated. The requirement of orthogonality places serious limitations on the Hebbian Learning Rule. A more powerful learning rule is the delta rule, which utilizes the discrepancy between the desired and actual output of each output unit to change the weights feeding into it.

Wijnew = Wijold  + α fi  (aiq)gj(pjp)

Wijnew = Wijold  + α aiq pjp

******

## Pseudoinverse Rule:

When the prototype input patterns are not orthogonal. Hebb rule produces some error. Then we use Pseudoinverse rule to minimize error in pattern recognition.





## Auto Associative Memory:

In Auto Associative memory the desired output vector is equal to the input vector. We will use an auto associative memory to to store a set of patterns and then to recall these patterns, even when corrupted patterns are provided as inputs. The patterns we want to store are “0” ,”1 “ and “2”.

As we are programming an auto associative memory, these patterns represent the input vectors and targets on the initial training session. The patterns are presented in a 6\*5 grid. On the testing session we will use deformed patterns as inputs and get the output presenting which one is it similar with.

At first, we will convert the matrix into vectors, and make them prototype patterns for the network. The white squares are presented as 1 and black as -1.



**Figure**: **Auto Associative network for pattern recognition**

C:\Users\Tanjina\Dropbox\Matlab code\Auto Associative Memory Verson 2\0.png

Fig: Training Pattern for ‘0’

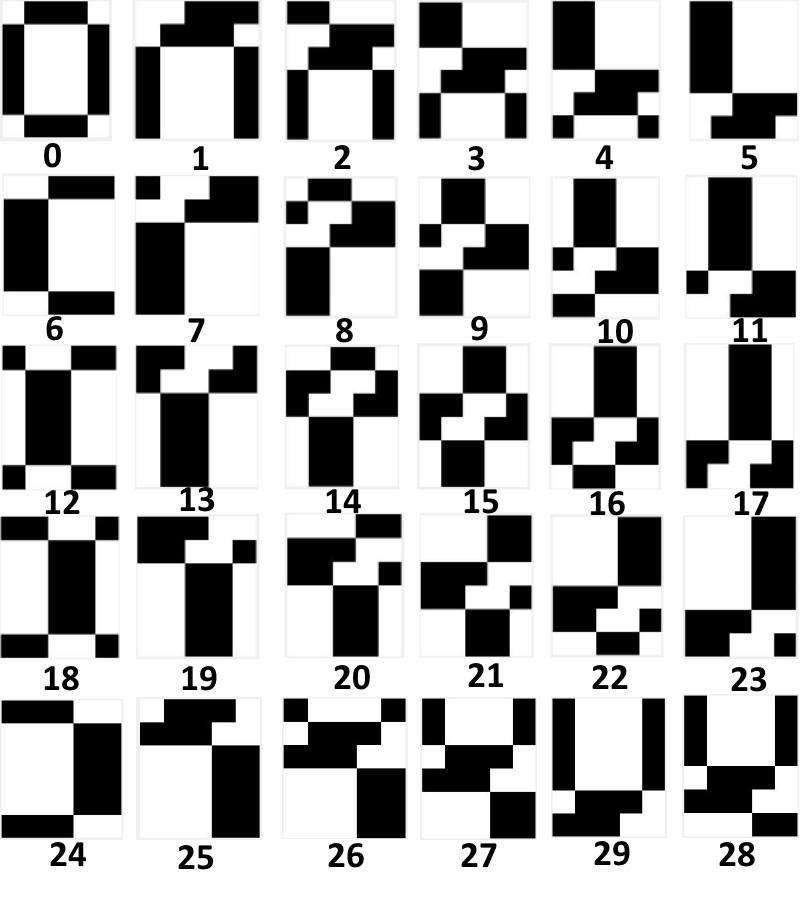
****

Figure: Deformed testing patterns for“0”

C:\Users\Tanjina\Dropbox\Matlab code\Auto Associative Memory Verson 2\1.png

Fig: Training Pattern for ‘1’

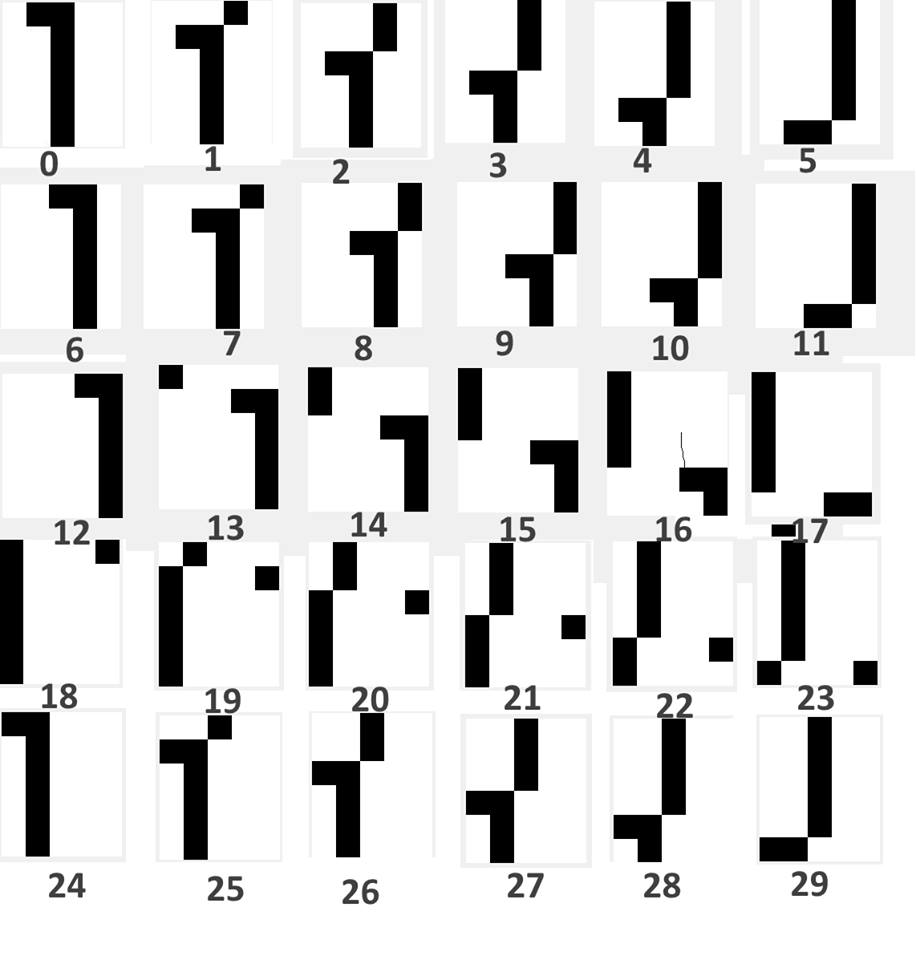
****

Figure: Deformed testing patterns for “1”



Fig: Training Pattern for ‘2’

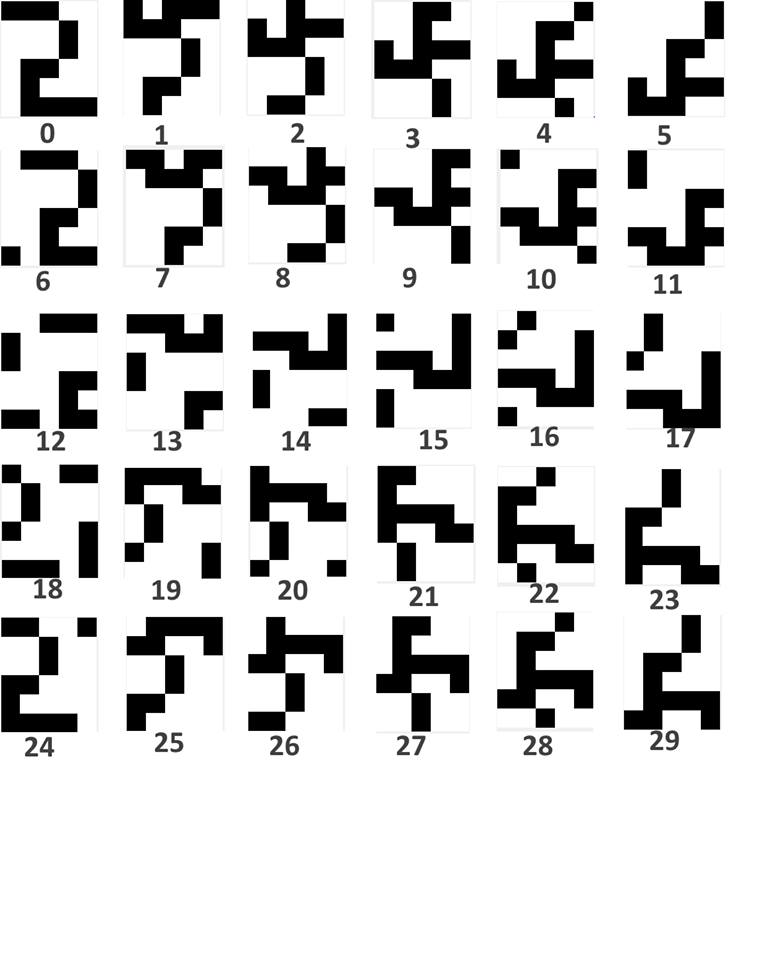
****

Figure: Deformed testing patterns for “2”

## Accuracy of Hebb’s rule:

From the pattern output that we get after applying Hebb’s rule are less efficient. The accuracy is quite less than that expected. The overall accuracy of Hebb’s rule is only 38.02%. For more convoluted patter the accuracy to recognize them decreases.

## Accuracy of Hebb’s rule:

From the pattern output that we get after applying Pseudoinverse rule are comparatively more efficient. The overall accuracy of Hebb’s rule is only 42.34%. However, for more convoluted patter the accuracy to recognize them decreases.

## Matlab code for Auto Associative Network:

% ……0…….

p1 = [ 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;

1 -1 -1 -1 1];

% …….1………

p2=[ 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;

1 1 -1 1 1];

% ……2………

p3 = [-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;

1 -1 -1 -1 -1 ];

PP= [p1,p2,p3];

fid=fopen('C:\Users\Tanjina\Dropbox\Matlab code\Auto Associative Memory Verson 2\errors.txt','w');

pnew=p2;

pold=pnew;

for n=1:1:29

pnew(n+1:30)=pold(1:30-n);

pnew(1:n)=pold(30-n+1:30);

imtool(pnew);

a = = AAME\_PseusoInv(p1,p2,p3,PT);

b1 = reshape(pnew,1,30);

fprintf(fid, ' %d \n',b1);

fprintf(fid,'[{(');

fprintf(fid, ' %d \n',a);

fprintf(fid,')}]');

end

fclose(fid);

## Auto Associative Network using Hebbian rule:

function [ a ] = AAM\_Hebbian( p1,p2,p3,PT )

figure,imshow(p1)

figure,imshow(p2)

figure,imshow(p3)

p1 = reshape(p1,30,1);

p2 = reshape(p2,30,1);

p3 = reshape(p3,30,1);

p = [p1 p2 p3];

% T = [t1; t2; t4]'

% T = [t1; t2; t3; t4]'

pp = inv(p' \* p) \* p'

w = p\*p’

PT = reshape(PT, 30,1)

w = p \* pp

a = hardlims( w\*PT)

end

## Auto Associative Network using Pseudoinverse rule:

function [ a ] = AAME\_PseusoInv(p1,p2,p3,PT)

% p1 = [ 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;

% 1 -1 -1 -1 1];

% % Zero

%

% p2=[ 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;

% 1 1 -1 1 1]; % one

%

% p3 = [-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;

% 1 -1 -1 -1 -1 ]; % two

figure,imshow(p1)

figure,imshow(p2)

figure,imshow(p3)

p1 = reshape(p1,30,1);

p2 = reshape(p2,30,1);

p3 = reshape(p3,30,1);

p = [p1 p2 p3];

% T = [t1; t2; t4]'

% T = [t1; t2; t3; t4]'

pp = inv(p' \* p) \* p'

PT = reshape(PT, 30,1)

w = p \* pp

a = hardlims( w\*PT)

end

# Topic 03: Backpropagation

Backpropagation, an abbreviation for "backward propagation of errors", is a common method of training artificial neural networks used in conjunction with an optimization method such as gradient descent. The method calculates the gradient of a loss function with respect to all the weights in the network. The gradient is fed to the optimization method which in turn uses it to update the weights, in an attempt to minimize the loss function.

Backpropagation requires a known, desired output for each input value in order to calculate the loss function gradient. It is therefore usually considered to be a supervised learning method, although it is also used in some unsupervised networks such as auto encoders. It is a generalization of the delta rule to multi-layered feedforward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Backpropagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.

Feed Forward




Backpropagation





Steepest descent Weight Update





## Matlab code for Backpropagation:

w1 = rand(2,1);

b1 = rand(2,1);

w2 = rand(1,2);

b2 = rand(1,1);

a0 =1;

p=1;

l = 0.8;

i =0 ;

while (true)

a1 = logsig (w1\*a0+b1);

a2 = purelin(w2\*a1+b2);

e = (1+sin(pi\*p/4)) - a2;

s2 = -2\*1\* e; %...............sensitivity calculation….

jj1= (1-a1(1,1))\*a1(1,1);

jj2 = (1-a1(2,1))\*a1(2,1);

k = [jj1 0;0 jj2];

s1 = k\*w2'\*s2;

%...... weight update………….

nw2= w2 - l\*s2\*a1';

nb2 = b2 - l\*s2;

nw1 = w1 - l\*s1\*a0';

nb1 = b1 - l\*s1;

h = nw2- w2;

g = sum(h);

if (g==0)

break;

end

w2 = nw2;

b2 = nb2;

w1= nw1;

b1= nb1; i = i +1;

end

## Function Approximation:

w1 = rand(2,1);

b1 = rand(2,1);

w2 = rand(1,2);

b2 = rand(1,1);

alpha = 0.1;

p = -1;

t = 1 + sin((pi/4)\*p);

w1n = [0;0];

w2n = [0 0];

c = 0;

while true

a1 = logsig(w1\*p + b1);

a2 = purelin(w2\*a1 + b2);

e = t - a2;

s2 = -2\*1\*e;

s1 = [(1-a1(1,1))\*a1(1,1) 0; 0 (1-a1(2,1))\*a1(2,1)]\*w2'\*s2;

w2n = w2 - alpha\*s2\*a1';

b2n = b2 - alpha\*s2;

w1n = w1 - alpha\*s1\*p;

b1n = b1 - alpha\*s1;

if w1 == w1n

if w2 == w2n

break;

end

end

w1 = w1n;

b1 = b1n;

w2 = w2n;

b2 = b2n;

c = c+1;

end

# Topic 04: Competitive Network

## Matlab code for competitive network:

p1 = [-0.1961; 0.9806];

p2= [0.1961; 0.9806];

p3=[0.9806;0.1961];

p4= [0.9806;-0.1961];

p5= [-0.5812;-0.8137];

p6=[-0.8137;-0.5812];

w1 = [0.7071; -0.7071];

w2 = [0.7071; 0.7071];

w3 = [-1.0000;0.0000];

% W = [w1'; w2'; w3'];

po = [p1,p2,p3,p4,p5,p6];

po = po';

i=1;

c=0;

while true

W = [w1'; w2'; w3'];

p = (po(i,:))';

a = compet(W\*p);

if a(1,1) == 1

nw = w1 + 0.8\*(p-w1);

if w1 == nw

break;

end

w1 = nw;

elseif a(2,1) == 1

nw = w2 + 0.8\*(p-w2);

if w2 == nw

break;

end

w2 = nw;

elseif a(3,1) == 1

nw = w3 + 0.8\*(p-w3);

if w3 == nw

break;

end

w3 = nw;

end

i = i+1;

if i>6

i=1;

end

c = c+1

if c == 50000

break;

end

end