ECE 559 – Neural Networks

Character Recognition using Counter Propagation Network

Assignment – III

Arindam Bose
UIN: 665387232
10-8-2015

1. *Problem Statement*

Use Counter Propagation network (CPNN) for character recognition to recognize 3 characters of your choice (for example, ‘A’, ‘C’, and ‘N’). Use 8x8 grid to define your characters. The network should also be able to identify 'other' character i.e. any character other than the ones used for training should be classified as 'other'. So, in all, your CPNN should be able to classify any 8x8 character into one of the four groups (‘A’, ‘C’, ‘N’ or ‘other’).

Test your network for characters with different bit noise.

Also, test it for characters other than the 3 characters used for training.

2. *Introduction*

The Counter Propagation network combines the Self-Organizing (Instar) networks of Kohonen and the Grossberg’s Oustar net consisting of one layer of each. It has good properties of Generalization that allow it to deal well with partially incomplete or partially incorrect input vectors. Counter Propagation network serves as a very fast clustering network.

a. Kohonen Self-Organizing Map Layer: The Kohonen layer is a “Winner-take-all” (WTA) layer. Thus, for a given input vector, only one Kohonen layer output is 1 whereas all others are 0. No training vector is required to achieve this performance. Hence, the name: Self-Organizing Map Layer (SOM-Layer).

Let the net output of a Kohonen layer neuron be denoted as k_j. Then

$$k_j = \sum_{i=1}^{m} w_{ij} x_i = \mathbf{w}_j^T \mathbf{x};$$

$$\mathbf{w}_j \triangleq \left[w_{1j} \dots w_{mj}\right]^T$$

$$\mathbf{x} \triangleq [x_1 \dots x_m]^T$$

where $j = 1, 2, \dots p, p$ being the number of different patterns (classes) considered, and where m is the dimension of the input Vector. Subsequently, for the hth $(j = h)$ neuron, where $k_h > k_{j \neq h}$. We then set w_j such that:

$$k_h = \sum_{i=1}^{m} w_{ih} x_i = 1 = \mathbf{w}_h^T \mathbf{x}$$

and, possibly via lateral inhibition $k_{j \neq h} = 0$

b. Grossberg Layer: The input of the Grossberg layer is the weighted output of the Kohonen layers. The number of neurons (p) in Grossberg Layer must be at least that of the binary representation of the number of different classes (p) to be classified by the CP network.

Denoting the net output of the Grossberg layer as g_q then

$$g_q = \sum_i k_i v_{iq} = \mathbf{k}^T \mathbf{v}_q;$$
$$\mathbf{k} \triangleq \left[k_1 \dots k_p\right]^T$$
$$\mathbf{v}_q \triangleq \left[v_{1q} \dots v_{pq}\right]^T$$

where $q = 1,2,\dots,r, r$ being equal to the dimension of the binary representation of p, such that, $p = 7$ (binary: 111) yields $r = 3$, etc...

Furthermore, by the "winner-take-all" nature of the Kohonen layer; if

$$\begin{cases} k_h = 1 \\ k_{j\neq h} = 0 \end{cases}$$

then,

$$g_q = \sum_i k_{ij} v_{jq} = \mathbf{k}^T \mathbf{v}_q = k_h v_{hq} = v_{hq}$$

3. Description of Design of the CNN

The goal of this case study is to recognize three characters 'A', 'C', 'N'. To do this, two-layer Hopfield network is created, it is trained with standard data sets (8×8); make the algorithm converge and it is tested the network with a set of test data other than trained set and also trained data having varying bit noise.

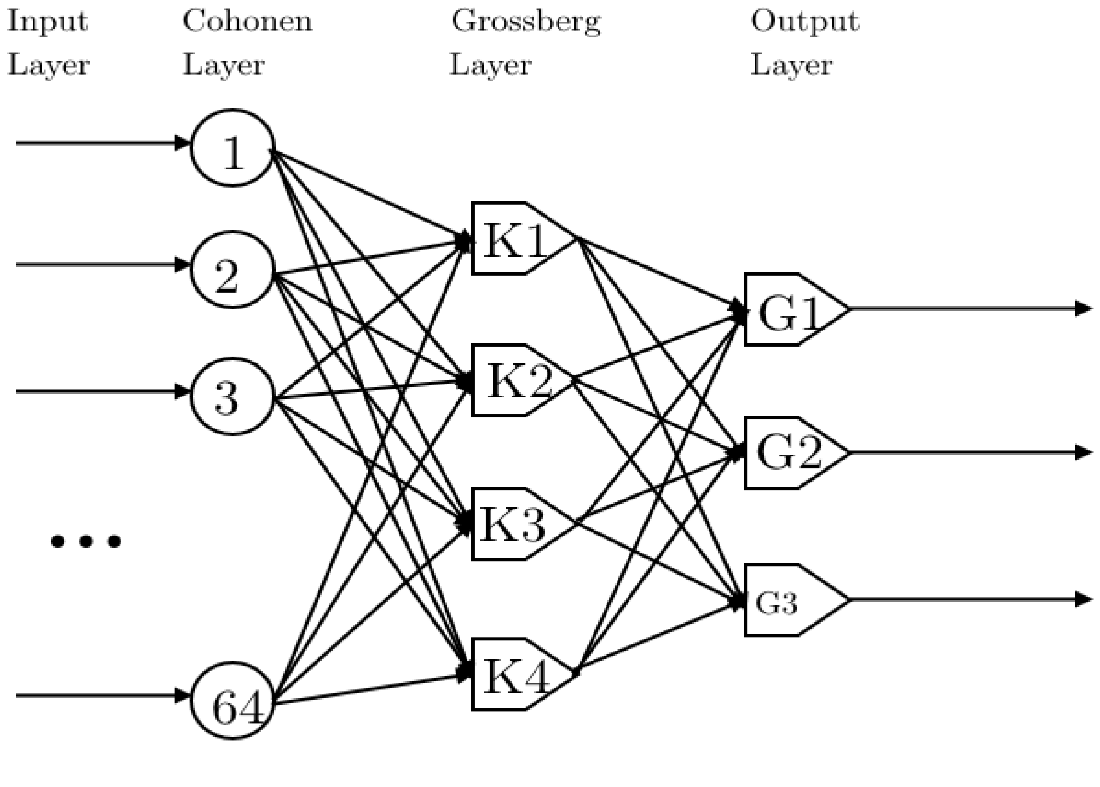

Fig. 1. Schematic design of the Hopfield neural network

a. Network Structure: There are 64 inputs to the network. So 64 neurons are used to form the input layer as depicted in Fig. 1

b. Dataset Design: In this CPNN we teach the network to recognize characters 'A', 'C' and 'N'. To train the network to produce error signal we will use another 1 character 'X'. We are interested in checking the response of the network to errors on the characters which were not involved in the training procedure. The characters to be recognized are given on an 8×8 grid. Each of the 64 pixels is set to either -1 (white pixels) or 1 (black pixels) as in Fig. 2.

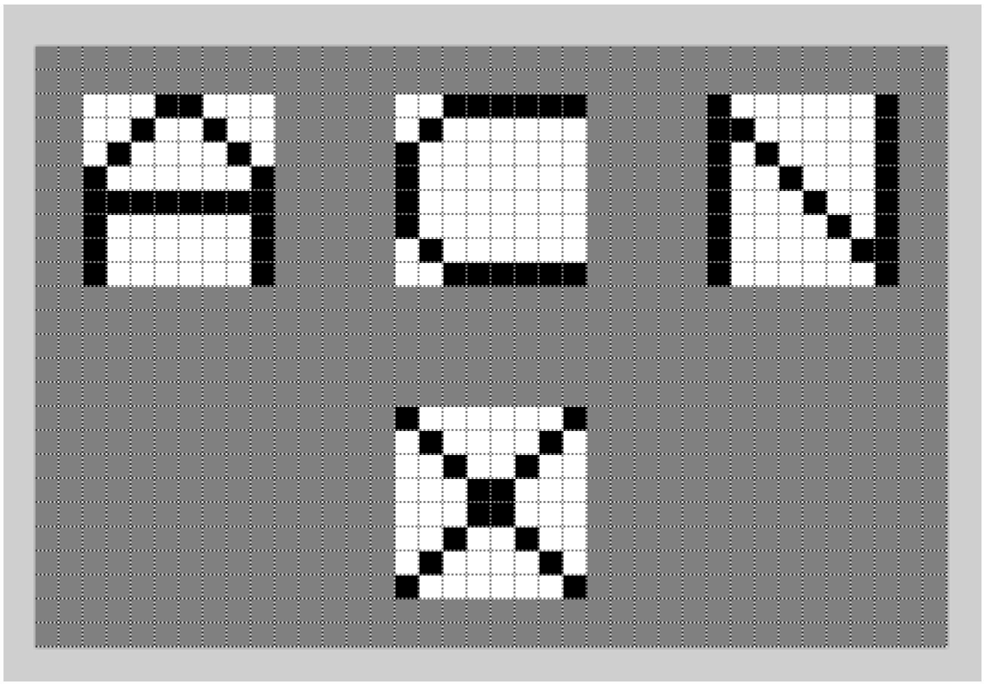

Fig. 2. Dataset: 'A', 'C', 'N', 'X'

4. Training

a. Training of the Kohonen Layer: The Kohonen layer acts as a classifier where all similar input vectors, namely those belonging to the same class produce a unity output in the same Kohonen neuron. Subsequently, the Grossberg layer produces the desired output for the given class as has been classified in the Kohonen layer above. In this manner, generalization is then accomplished.
It is usually required to normalize the Kohonen layer's inputs, as follows

$$x_i' = \frac{x_i}{\sqrt{\sum_j x_j^2}}$$

yield a normalized input vector x' where, $(\mathrm{x}')^T\mathrm{x}' = 1 = \|\mathrm{x}'\|$

The training of the Kohonen layer now proceeds as follows:

1. Normalize the input vector x to obtain x'.
2. The Kohonen layer neuron whose

$$(\mathrm{x}')^T \mathrm{w}_h = k'_h$$

is the highest, is declared the winner and its weights are adjusted to yield a unity output $k'_h = 1$.

Whereas in practically all NN's the initial weights are selected to be of pseudo random low values, in the case of Kohonen networks, any pseudo random weights must be normalized if an approximation to x' is to be of any meaning. But then even normalized random weights may be too far off from x' to have a chance for convergence at a reasonable rate. Furthermore if there are several relatively close classes that are to be separated via Kohonen network classification, one may never get there. If, however, a given class has a wide spread of values, several Kohonen neurons may be activated for the same class. Still, the latter situation can be subsequently corrected by the Grossberg layer which will then guide certain different Kohonen layer outputs to the same overall output.

b. Training of the Grossberg Layer: First the outputs of the Grossberg layer are calculated as in other networks, namely

$$g_i = \sum_j k_j v_{ij} = k_h v_{ih} = v_{ih}$$

k_j being the Kohonen layer outputs and v_{ij} denoting the Grossberg layer weights. Obviously, only weights from non-zero Kohonen neurons (non-zero Grossberg layer inputs) are adjusted. Weight adjustment follows the relations often used before, namely:

$$v_{ij}(n+1) = v_{ij}(n) + \beta[T_i - v_{ij}(n)k_j]$$

T_i being the desired outputs (targets), and for the $n+1$ iteration β being initially set to about 1 and is gradually reduced. Initially v_{ij} are randomly set to yield a vector of norm 1 per each neuron. Hence, the weights will converge to the average value of the desired outputs to best match an input-output $(x - T)$ pair.

c. Setting of Weights:

1. Get all training data vectors $X_i, i = 1, 2 \dots L$
2. For each group of data vectors belonging to the same class, $X_i, i = 1, 2 \dots N$.
   a. Normalize each $X_i, i = 1, 2 \dots N, X'_i = \frac{X_i}{sqrt(\sum X^2 j)}$
   b. Compute the average vector $X = \frac{\Sigma X'_j}{N}$

c. Normalize the average vector $X, X' = \frac{X}{sqrt(X^2)}$

d. Set the corresponding Kohonen Neuron's weights $W_k = X$

e. Set the Grossberg weights $[W_{1k} W_{1k} \dots W_{1k}]$ to the output vector Y

3. Repeat step 2 until each class of training data is propagated into the network.

5. *Results*

a. Network Training: To train the network to recognize the above characters we applied the corresponding 8×8 grids in the form of 1×64 vectors to the input of the network. At this point the output of the last layer was as below:

```
Training set: 1: Pass
Training set: 2: Pass
Training set: 3: Pass
Training set: 4: Pass
```

Training set 1, 2, 3, 4 corresponds to letter 'A', 'C', 'N', 'X' respectively. The system successfully recognized all the trained exemplars.

b. Robustness: To investigate the robustness of the neural network, I added several noise bits to the input and got the following results. I tested with 1-50 bit noises:

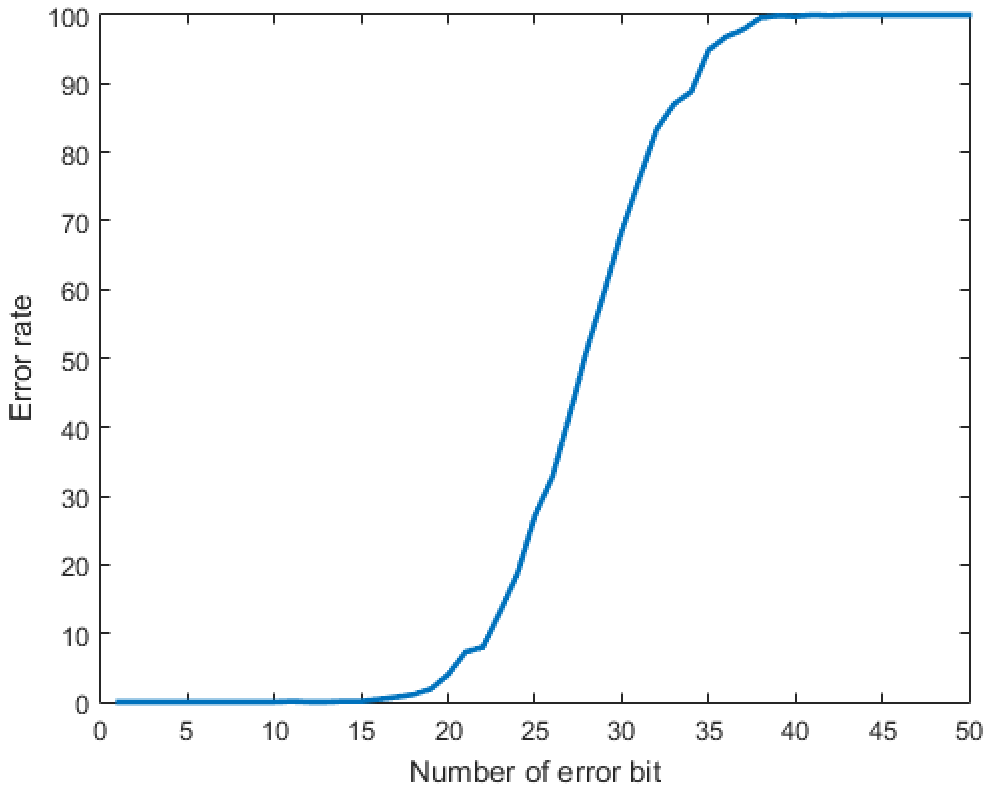

Fig. 3. Graph: Error vs Number of error-bit.

From the results it is evident that the network works fairly well even for 15-bit error. As the number of error bit increases from 20-bit, error rate also increases, but fairly slowly.

6. *Conclusions*

The CPNN discussed here performed quiet well with both noise and no-noise dataset. Also, it was able to detect all the False Input correctly. Also the training time with 4 different training exemplar was **0.0036789 seconds** which is very small. Testing time was also small, averaging **0.00799 seconds** for each character. The Counter propagation network is robust with high convergence rate and also it is seen that it has very high success rate even if in the case of large bit errors.

7. *References*

[1] Graupe D., "Principles of Artificial Neural Networks – 3ed," ch 8, pp. 185 - 201, 2013.

8. *Source Code (MATLAB R2013b)*

a. main.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% ECE 599 Neural Networs                                       %
% Name: Arindam Bose                                           %
% UIN: 665387232                                               %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Implementation of Character Recognition system using Counter
Propagation Neural Network
%% Initialization
Initialization();

%% Main menu
first = true;
while first
    clc;
    disp('------------- Main Menu -------------');
   disp('1: Train network');
   disp('2: Test with trained exemplars');
    disp('3: Test with noise');
    disp('4: Generate graph for error rates');
   disp('0: Exit');
    ch = input('Your choice: ', 's');
    switch ch
        case '1'
            tic;
            TrainNetwork(); % Train network
            t = toc;
            disp(['Training complete in ' num2str(t) ' sec']);
        case '2'
            tic;
            TestWithExemplars(); % Test with trained exemplars A, C,
N, others
            t = toc;
            disp(['Testing trained exemplar complete in ' num2str(t)
' sec']);
        case '3'
            n = input('Number of error bits ? ');
            RecognitionWithError(n); % Recognition with noise
        case '4'
            n = input('Number of final error bits ? ');
            GenerateGraph(n);
        case '0'
            first = false;
            disp('Exiting...');
    end
    disp('Press any key to continue...');
    pause;
end
clear all;
```

b. Initialization.m

```
function Initialization()
% Initialization of global variables and initial weights
%% Global variabls
global CPNetwork Exemplars NumberPerTrainingSet;

NoOfNeuron = 64;
weightRange = [-0.5 0.5];
NumberPerTrainingSet = 250;

%% Exemplar definition
Exemplars = [];
Exemplars(1).input = [ -1, -1, -1,  1,  1, -1, -1, -1; ... % 'Letter
A'
                       -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,  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];
Exemplars(1).name = 'A';
Exemplars(1).classID = 1;
Exemplars(1).output = [0 0 1];

Exemplars(2).input = [ -1, -1,  1,  1,  1,  1,  1,  1;  ... %
'Letter C'
                       -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, -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];
Exemplars(2).name = 'C';
Exemplars(2).classID = 2;
Exemplars(2).output = [0 1 0];

Exemplars(3).input = [  1, -1, -1, -1, -1, -1, -1, 1;  ... % 'Letter
N'
                        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, 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];
Exemplars(3).name = 'N';
Exemplars(3).classID = 3;
Exemplars(3).output = [1 0 0];

Exemplars(4).input = [  1, -1, -1, -1, -1, -1, -1,  1;  ... %
'Letter X'
                       -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, -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];
Exemplars(4).name = 'X';
Exemplars(4).classID = 4;
Exemplars(4).output = [0 0 0];

%% Counter Propagation network structure
CPNetwork = [];
CPNetwork.layerMatrix = [NoOfNeuron size(Exemplars,2)
size(Exemplars(1).output,2)];

%% Kohonen layer
kLayer.number = CPNetwork.layerMatrix(2);
kLayer.error = [];
kLayer.output = [];
kLayer.neurons = [];
kLayer.Z = [];
kLayer.weights = [];

% create a default layer
for i = 1: kLayer.number
   % create a default neuron
   neuron = [];
    offset = (weightRange(1) + weightRange(2))/2.0;
    range = abs(weightRange(2) - weightRange(1));
    weights = (rand(1, CPNetwork.layerMatrix(1)) - 0.5 )* range +
offset;

    neuron.weights = weights;
   neuron.weightsUpdateNumber = 0;
    neuron.z = 0;
    neuron.y = 0;
    kLayer.neurons = [kLayer.neurons, neuron];
    kLayer.weights = [kLayer.weights; weights];
end
CPNetwork.kLayer = kLayer;

%% Grossberg Layer
gLayer.number = CPNetwork.layerMatrix(3);
gLayer.error = [];
gLayer.output = [];
gLayer.neurons = [];
gLayer.Z = [];
gLayer.weights = [];
% create a default layer
for ind = 1: gLayer.number
   % create a default neuron
   neuron = [];
    offset = (weightRange(1) + weightRange(2))/2.0;
    range = abs(weightRange(2) - weightRange(1));
    weights = (rand(1, CPNetwork.layerMatrix(2)) - 0.5 )* range +
offset;

    neuron.weights = weights;
    neuron.weightsUpdateNumber = 0;
    neuron.z = 0;
    neuron.y = 0;
    gLayer.neurons = [gLayer.neurons, neuron];
    gLayer.weights = [gLayer.weights; weights];
end
CPNetwork.gLayer = gLayer;
end
```

c. TrainNetwork.m

```
function TrainNetwork()
% Function for training network with training exemplars
global CPNetwork Exemplars;

eSize = size(Exemplars);
kWeights = [];
gWeights = zeros(CPNetwork.gLayer.number, CPNetwork.kLayer.number);
for i = 1 : eSize(2) % Count training exemplars and applying weights
    mIn = Exemplars(i).input(:);
   mOut = Exemplars(i).output(:);
   mID = Exemplars(i).classID;
   mIn = mIn / sqrt(sum(mIn.*mIn));

   % training the Kohonen Layer
    prevWeights = CPNetwork.kLayer.neurons(mID).weights;
    weightUpdateNumber =
CPNetwork.kLayer.neurons(mID).weightsUpdateNumber + 1;
    if weightUpdateNumber >= 1
```

```
        mIn = (prevWeights' * weightUpdateNumber + mIn) /
weightUpdateNumber;
        mIn = mIn / sqrt(sum(mIn.*mIn));
    end
    CPNetwork.kLayer.neurons(mID).weights = mIn';
    CPNetwork.kLayer.neurons(mID).weightsUpdateNumber =
weightUpdateNumber;
    kWeights = [kWeights; mIn'];

   % training the Grossberg Layer
    if weightUpdateNumber >= 1
        mOut = (mOut * weightUpdateNumber + mOut) /
weightUpdateNumber;
    end
    gWeights(:, mID) = mOut;
end

for i = 1: CPNetwork.gLayer.number % store weights
    CPNetwork.gLayer.neurons(i).weights = gWeights(i,:);
end
CPNetwork.kLayer.weights = kWeights;
CPNetwork.gLayer.weights = gWeights;
end
```

d. <u>TestWithExemplars.m</u>

```
function TestWithExemplars()
% funtion for test with trained/untrained exemplar
global Exemplars;

str = [];
eSize = size(Exemplars);
for i = 1 : eSize(2) % Count exemplar
    Propagation(Exemplars(i).input);
    [output] = Classification();
    if strcmp(output.Name, Exemplars(i).name)
        astr = ['Training set: ', num2str(i), ': Pass: Error: ',
num2str(output.Error)];
    else
        astr = ['Training set: ', num2str(i), ': Untrained'];
    end
    str = char(str, astr);
end
disp(str);
end
```

e. <u>Propagation.m</u>

```
function Propagation(inputData)
% Function for propagating the Hopfield
global CPNetwork;
```

```
%% get inputs
nnInputs = inputData(:);

%% propagation of Kohonen Layer
zOut = CPNetwork.kLayer.weights * nnInputs;
[~, zMaxInd] = max(zOut);
yOut = zeros(size(zOut));
yOut(zMaxInd) = 1;
CPNetwork.kLayer.Z = zOut;
CPNetwork.kLayer.output = yOut;
for i = 1 : CPNetwork.kLayer.number
    CPNetwork.kLayer.neurons(i).z = zOut(i);
    CPNetwork.kLayer.neurons(i).y = yOut(i);
end

%% propagation of Grossberg Layer
zOut = CPNetwork.gLayer.weights * yOut;
yOut = zOut;
CPNetwork.gLayer.analogOutput = zOut;
CPNetwork.gLayer.output = yOut;
for i = 1 : CPNetwork.gLayer.number
    CPNetwork.gLayer.neurons(i).z = zOut(i);
    CPNetwork.gLayer.neurons(i).y = yOut(i);
end
end
```

f. Classification.m

```
function [output] = Classification()
% Function for Hopfield classification
global CPNetwork Exemplars;

delta = [];
eSize = size(Exemplars);
outputG = CPNetwork.gLayer.output;
for i = 1 : eSize(2) % Count training exemplars
    aSet = Exemplars(i).output(:);
    diff = abs(aSet - outputG);
    diff = diff.^2;
    newError = sum(diff);
    delta = [delta, newError];
end
[eMin, eInd] = min(delta);
output.Name = Exemplars(eInd).name;
output.Vector = Exemplars(eInd).output;
output.ClassID = Exemplars(eInd).classID;
output.Error = eMin;
end
```

g. RecognitionWithError.m

```
function [errorrate] = RecognitionWithError(nBitError)
% Function for recognition with error

testData = GenerateTestData(nBitError);
tSize = size(testData);
str = [];
success = 0;
for i = 1: tSize(2) % count number of testdata
    Propagation(testData(i).input);
    [output] = Classification();
    strFormat = ' ';
    vstr = char(strFormat,num2str(i));
    if strcmp(output.Name, testData(i).name)
        success = success + 1;
        astr = [vstr(2,:), ': Pass -> Error: ',
num2str(output.Error)];
    else
        astr = [vstr(2,:), ': Fail'];
    end
    str = char(str, astr);
end
errorrate = 100 - (success*100/tSize(2));
astr = ['Error rate: ', num2str(errorrate),'%'];
str = char(str, astr);
disp(str);
end
```

h. GenerateTestData.m

```
function testData = GenerateTestData(nBitError)
% Function for generating test inputs with error bit
global Exemplars NumberPerTrainingSet;

testData = [];
eSize = size(Exemplars);
id = 1;
for i = 1 : eSize(2) % Count training exemplars
    input = Exemplars(i).input;
    name = Exemplars(i).name;
   output = Exemplars(i).output;
   classID = Exemplars(i).classID;
    iSize = size(input);
    for j = 1: NumberPerTrainingSet
        row = [];
        col = [];
        flag = ones(iSize);
        bitErrorNum = 0;
        while bitErrorNum < nBitError
            x = ceil(rand(1) * iSize(1));
            y = ceil(rand(1) * iSize(2));
            if x <= 0
                x = 1;
            end
            if y <= 0
                y = 1;
            end
            if flag(x, y) ~= -1
                bitErrorNum = bitErrorNum + 1;
                flag(x, y) = -1;
                row = [row, x];
                col = [col, y];
            end
        end
        newInput = input;
        for p = 1:nBitError
            newInput(row(p), col(p)) = newInput(row(p), col(p)) * (-
1);
        end
        testData(id).input = newInput;
        testData(id).name = name;
         testData(id).output = output;
         testData(id).classID = classID;
        id = id + 1;
    end
end
```

i. <u>GenerateGraph.m</u>

```
function GenerateGraph(nBitError)
% Function for generating Error rate vs Number of error bit graph
for i = 1 : nBitError
    errorate(i) = RecognitionWithError(i);
end
figure;
plot(errorate, 'LineWidth', 2);
xlabel('Number of error bit');
ylabel('Error rate');
end
```