RAJSHAHI UNIVERSITY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Course Title: Sessional Based on CSE 4203
Course No: CSE 4204

LAB REPORT

Submitted by:	Submitted to:
Name: Md. Mehedi Hasan Roll:143036 Section: A Date: 17/04/2019	Prof. Dr. Md. Rabiul Islam Head Dept. of CSE, RUET

TABLE OF CONTENT

Lab 1

Title: Nearest Neighbor Algorithm

Objectives:

- Classify unknown patterns
- To make decision based on the shortest distance to the neighbouring class samples
- For handling rogue pattern, average distance is used instead of minimum distance

Theory:

The idea of Nearest Neighbour algorithm is simple. When an unknown data is presented to classify, the algorithm first finds the point from all the classes that are nearest from the data point.

$$(X) = closest(class1) - closest(class2)$$

Then find the point which has lowest distance among these data points. Distance is measured using Euclidean distance which is defined by the following equation:

$$d(X,Y)_{euc} = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

There is an anomaly which is known as rogue pattern when a point is misclassified. Then all point near that point will also be misclassified as they are near that point even they should belong to other class. To avoid this anomaly, we use average distance instead of minimum distance from all the points of the classes. So, the effect of rogue points reduces and we get accurate results.

Methodology:

Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. For i=0 to m:

   Calculate Euclidean Distance d(arr[i], p).
3. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
4. Return the majority label among S.

Code:

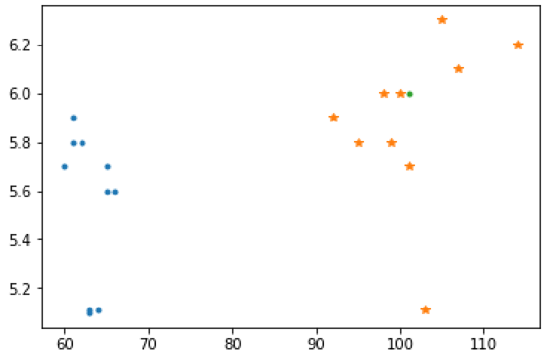
```
#import needed packages
import numpy as np
import matplotlib.pyplot as plt
import csv

def loadDataSet(path):
   data = np.loadtxt(path,
delimiter=',',skiprows=1)  #load csv
file without the header row
   sorted_data = sorted(data[:], key =
lambda x: x[2]) #sort the data based
on 3rd column i.e. class
   class1 = []
   class2 = []
   for i in range(len(sorted_data)):
      if(sorted_data[i][2] == 0):

class1.append([sorted_data[i][0],
sorted_data[i][1]]) #class 1 contains
data with class label 0
      else:
       class2.append([sorted_data[i][0],
sorted_data[i][1]]) #class 2 contains
data with class label 1
   return (np.array(class1),
np.array(class2))

def euclid_distance(x, y):
   return np.sqrt(np.sum((x-y)**2))

def NearestNeighbor(weight, height):
   global class1
   global class2
   len1 = len(class1)
   len2 = len(class2)

   min_distance_from_class1 =
999999
   for i in range(len1): #find the
minimum euclidean distance data
point from given point to every point
of class 1
      dist1 = euclid_distance(class1[i],
np.array((weight, height)))
      if(min_distance_from_class1 >
dist1):
         min_distance_from_class1 =
dist1

   min_distance_from_class2 =
999999
   for i in range(len2): #find the
minimum euclidean distance data
point from given point to every point
of class 2
      dist2 = euclid_distance(class2[i],
np.array((weight, height)))
      if(min_distance_from_class2 >
dist2):
         min_distance_from_class2 =
dist2

   if(min_distance_from_class1 <
min_distance_from_class2): #if the
point is closer to class 1
      class1 = np.vstack((class1,
np.array((weight, height)))) #append
that unknown point to class 1
      fig = plt.figure()
      plt.plot(class1[:, 0], class1[:, 1], 'x')
      plt.plot(class2[:, 0], class2[:, 1], '*')
      plt.show()
      return 1
   else:
      class2 = np.vstack((class2,
np.array((weight, height)))) #append
that unknown point to class 2
      fig = plt.figure()
      plt.plot(class1[:, 0], class1[:, 1], 'x')
      plt.plot(class2[:, 0], class2[:, 1], '*')
      plt.show()
      return 2

if __name__=="__main__":
   while(1):
      decision = input('Do you want to
proceed? Press y/n')
      if (decision == 'n' or decision ==
'N'):
         break
      weight = float(input('Enter
Weight:'))
```

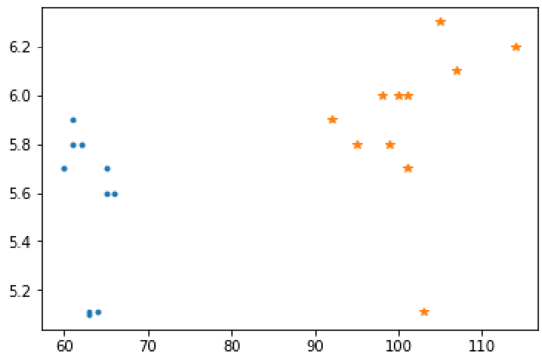
```
    height = float(input('Enter Height:'))
    path = 'NNData.csv'
    (class1, class2) = loadDataSet(path)
    fig = plt.figure()
    plt.plot(class1[:, 0], class1[:, 1], 'x')
    plt.plot(class2[:, 0], class2[:, 1], '*')
    plt.plot(weight, height, '.')
    plt.show()
    x = NearestNeighbor(weight, height)
    print('Predicted Class is:'+str(x))
    with open(path, 'a', newline='') as newFile:
        newFileWriter = csv.writer(newFile)

newFileWriter.writerow([weight, height, x-1])
```

Input:

Enter Weight:101
Enter Height:6.0

Output:





Predicted Class is:2

Code for handling rogue pattern:

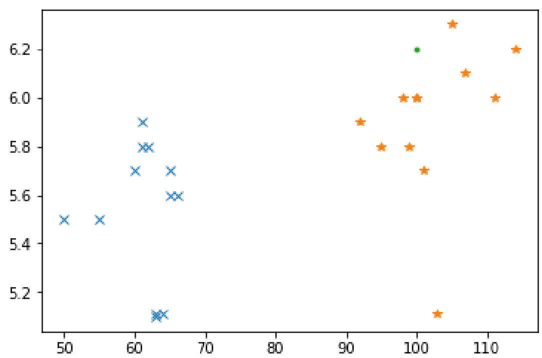
```
import numpy as np
import matplotlib.pyplot as plt
import csv

def loadDataSet(path):
   data = np.loadtxt(path, delimiter=',',skiprows=1)
   sorted_data = sorted(data[:], key = lambda x: x[2])
   class1 = []
   class2 = []
   for i in range(len(sorted_data)):
      if(sorted_data[i][2] == 0):
         class1.append([sorted_data[i][0], sorted_data[i][1]])
      else:
         class2.append([sorted_data[i][0], sorted_data[i][1]])
   return (np.array(class1), np.array(class2))

def euclid_distance(x, y):
   return np.sqrt(np.sum((x-y)**2))

def NearestNeighbor(weight, height):
   global class1
   global class2
   len1 = len(class1)
   len2 = len(class2)

   distance_from_class1 = 0
   for i in range(len1):
      distance_from_class1 += euclid_distance(class1[i], np.array((weight, height)))

   distance_from_class2 = 0
   for i in range(len2):
      distance_from_class2 += euclid_distance(class2[i], np.array((weight, height)))
```

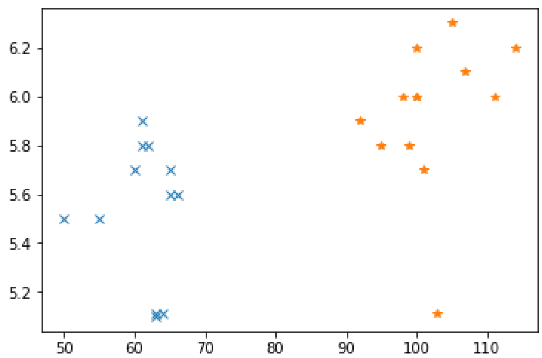
```
    if(distance_from_class1/len1 <
distance_from_class2/len2):
        class1 = np.vstack((class1,
np.array((weight, height))))
        fig = plt.figure()
        plt.plot(class1[:, 0], class1[:, 1], 'x')
        plt.plot(class2[:, 0], class2[:, 1], '*')
        plt.show()
        return 1
    else:
        class2 = np.vstack((class2,
np.array((weight, height))))
        fig = plt.figure()
        plt.plot(class1[:, 0], class1[:, 1], 'x')
        plt.plot(class2[:, 0], class2[:, 1], '*')
        plt.show()
        return 2
if __name__=="__main__":
    while(1):
        decision = input('Do you want to
proceed? Press y/n')
        if (decision == 'n' or decision ==
'N'):
            break
        weight = float(input('Enter
Weight:'))
        height = float(input('Enter
Height:'))
        path = 'NNData.csv'
        (class1, class2) =
loadDataSet(path)
        fig = plt.figure()
        plt.plot(class1[:, 0], class1[:, 1], 'x')
        plt.plot(class2[:, 0], class2[:, 1], '*')
        plt.plot(weight, height, '.')
        plt.show()
        x = NearestNeighbor(weight,
height)
        print('Predicted Class is:'+str(x))
        with open(path, 'a', newline='') as
newFile:
            newFileWriter =
csv.writer(newFile)

newFileWriter.writerow([weight,
height, x-1])
```

Input:

Enter Weight:100
Enter Height:6.2

Output:





Predicted Class is:2

Performance Analysis & Discussion:

If we use general nearest neighbour, it will be affected by outliers. It will produce incorrect predictions. If we use average distance instead of minimum distance, we would be able to solve the problem. Because averaging all the distance will give us more near results to actual mean without the outlier present

Lab 2

Title: Single Layer Perceptron Algorithm

Objectives:

- Understanding the basic of biological neuron
- Design a single layer perceptron that can classify binary data pattern

Theory:

A single layer perceptron is like a biological neuron that accumulates stimulation from dendrites and fires if the combined value is greater than the threshold value. There acts some weight factors in that accumulation process. If the weight of the stimulation is greater, it has higher impact on the neuron, if not, it has less impact on the output.

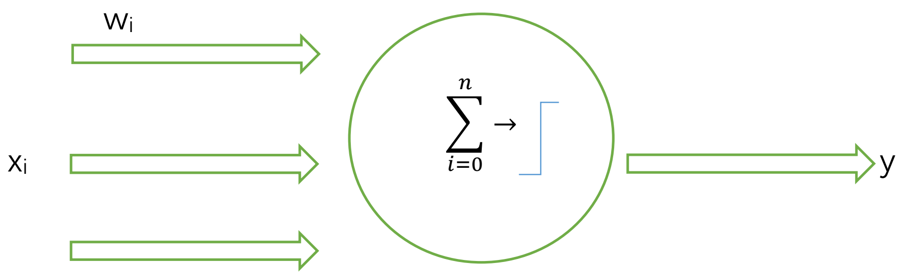


Fig 1: Single Layer Perceptron

Methodology:

1. Initialise weights and threshold
2. Present input and desired output
3. Calculate actual output

$$y(t) = f_h[\sum_{i=0}^{n} w_i(t)x_i(t)]$$

4. Adapt weights

$$\Delta = d(t) - y(t)$$
$$W_i(t + 1) = w_i + \eta \Delta x_i(t)$$

Where $0 \leq \eta \leq 1$, a positive gain term that controls adaptation rate.

Code:

```
#Code for perceptron learning
#formula => Wi = Wi + ita*delta*Xi
import numpy as np
import matplotlib.pyplot as plt
from numpy import binary_repr
import time
np.random.seed(7) #for
reproducibility

def genData(input_line):
   print('Input Lines: ' + str(input_line))
# no of bits in the input, 101 has 3 bits
so 3 input line in perceptron
   number_of_elements =
pow(2,input_line) #total combination

   # Making inputs
   inputs = []
   classes = []

   i = 0
   for i in
range(0,number_of_elements):
      tmp =
list(binary_repr(i,input_line))
#params=value, length, this returns
binary patterns for value of length
      if tmp[0]=='0': #if MSB is zero
         classes.append('0') #class label
      else:
         classes.append('1')
      inputs.append(tmp)
      i = i+1
   inputs = np.array(inputs)
   classes = np.array(classes)
   return (inputs.astype(np.float),
classes.astype(np.int)) #convert string
into float and int

def split_dataset(inputs, classes,
ratio): #split data with train data
ratio=ratio
   """
   split of the data will be like this:
   lets say total data 16, ratio = 0.8
   train_x  will be [0, 5) and [8, 13)
   """
   no = round((len(inputs) * ratio)/2)
   no_test = round(len(inputs)/2) - no
   train_x= []
   train_y= []
   test_x= []
   test_y= []
   for i in range(0, no):
      train_x.append(inputs[i])
      train_y.append(classes[i])
   for i in range(no, no+no_test):
      test_x.append(inputs[i])
      test_y.append(classes[i])
   for i in range(round(len(inputs)/2),
no+round(len(inputs)/2)):
      train_x.append(inputs[i])
      train_y.append(classes[i])
   for i in
range(no+round(len(inputs)/2),
len(inputs)):
      test_x.append(inputs[i])
      test_y.append(classes[i])

   return (train_x, train_y, test_x,
test_y)


def predict(input_x, weights,
threshold):
   #cumsum input_x with weights
and comparing with threshold value
   if float(np.dot(input_x, weights.T)) >
threshold:
      return 1
   else:
      return 0


def train_perceptron(train_x, train_y,
ita, threshold):
   no_of_input = len(train_x) #total
no of data
   no_of_input_line = len(train_x[0])
#input line no i.e. no of bits
```

```
    weights =
np.random.rand(1,len(train_x[0]))
#initialize weight vector randomly
    print('Initial Weights',weights)
    epoch = 0
    i = 0 # index for training
    while(1):
        epoch = epoch + 1
        input_x = train_x[i] #take an
input
        input_y = train_y[i]
        print("Input_x:", input_x)
        print("Input_y:", input_y)
        predicted_y = predict(input_x,
weights, threshold) # predict the
output
        print("Predicted_y:", predicted_y)
        delta = input_y - predicted_y
        #print(delta)
        weights = weights +
np.multiply(ita * delta, input_x)
#update weight
        if delta != 0: #if wrong
prediction
            i = 0 #start from begining with
updated weight
            continue
        else:
            i = i + 1 #else continue
        if i == no_of_input: #if no error
occured with latest weights
            break #then stop training
    print('Final Weights', weights)
    print("Total Steps:"+str(epoch))
    return weights


def train_perceptron2(train_x, train_y,
ita, threshold):
    no_of_input = len(train_x) #total
no of data
    no_of_input_line = len(train_x[0])
#input line no i.e. no of bits

    weights =
np.random.rand(1,len(train_x[0]))
#initialize weight vector randomly
    print('Initial Weights',weights)
    epoch = 0
    i = 0 # index for training
    current_class = 0
    position1 = 0
    position2 = int(no_of_input/2)
    class_1 = False
    while(1):
        epoch = epoch + 1
        input_x = train_x[i] #take an
input
        input_y = train_y[i]
        print("\nInput_x:", input_x)
        print("Input_y:", input_y)
        predicted_y = predict(input_x,
weights, threshold) # predict the
output
        print("Predicted_y:", predicted_y)
        delta = input_y - predicted_y
        #print(delta)
        weights = weights +
np.multiply(ita * delta, input_x)
#update weight
        if delta != 0: #if wrong
prediction
            if current_class == 0:
                i = position1 #start from
begining with updated weight
                class_1 = False
            else:
                i = position2
                class_1 = False
            continue
        else:
            i = i + 1 #else continue
            if i == no_of_input and
class_1 == False:
                i = position1
            elif i == no_of_input and
class_1 == True:
                break
            elif i == position2: #if no error
occured with latest weights
                class_1 = True
                current_class = 1
    print('Final Weights', weights)
    print("Total Steps:"+str(epoch))
    return weights
```

```
def test_perceptron(weights,
threshold, test_x, test_y):
    no_of_data = len(test_x)
    TP = 0
    TN = 0
    FP = 0
    FN = 0

    for i in range(no_of_data):
        pred_y = predict(test_x[i],
weights, threshold)
        print("X:",test_x[i])
        print("Y:",test_y[i])
        print("Predicted:",pred_y)
        if pred_y == test_y[i] and pred_y
== 1:
            TP = TP + 1
        elif pred_y == test_y[i] and
pred_y == 0:
            TN = TN + 1
        elif pred_y != test_y[i] and
pred_y == 1:
            FP = FP + 1
        else:
            FN = FN + 1
    accuracy = (TP +
TN)/(TP+TN+FP+FN)
    print("Accuracy:"+str(accuracy))
    return accuracy


if __name__=="__main__":
    (inputs, classes) = genData(10)
    ans = []

    train_ratios = [0.9, 0.8, 0.7, 0.6, 0.5]
    for train_ratio in train_ratios:
        (train_x, train_y, test_x, test_y) =
split_dataset(inputs, classes,
train_ratio)
        ita = np.random.random()
        threshold = np.random.random()
        #weights =
np.random.rand(1,len(train_x[0]))
#initialize weight vector randomly

        starting_time1 = time.time()
        trained_weights =
train_perceptron(train_x, train_y, ita,
threshold)
        ending_time1 = time.time()

        starting_time2 = time.time()
        trained_weights2 =
train_perceptron2(train_x, train_y, ita,
threshold)
        ending_time2 = time.time()

        print('\nTesting\n')
        accuracy =
test_perceptron(trained_weights,
threshold, test_x, test_y)
        accuracy2 =
test_perceptron(trained_weights2,
threshold, test_x, test_y)
        #print('\nTime Required: ',
ending_time - starting_time)
        x =
str(train_ratio)+'\t'+str(round(1 -
train_ratio,
2))+'\t'+str(round(ending_time1 -
starting_time1,
2))+'\t'+str(round(ending_time2 -
starting_time2,
2))+'\t'+str(round(accuracy * 100,
2))+'\t'+str(round(accuracy2 *
100,2))+'\t\n'
        ans.append(x)

print('\t\tTime\t\tAccuracy')
print('\nLearn\tTest\tAlgo(1)\tAlgo(2)\t
Algo(1)\tAlgo(2)\n')
for i in ans:
    print(i)
```

Results:

If everything (weights, ita, k) is same:

		Time(in seconds)		Accuracy (in %)	
Learn(%)	Test(%)	Algo(1)	Algo(2)	(Algo(1)	Algo(2)
90	10	4.57	12.68	100.0	100.0
80	20	2.94	2.95	100.0	100.0
70	30	2.15	4.35	100.0	100.0
60	40	1.99	3.66	100.0	100.0
50	50	1.8	2.84	99.8	99.8

If only Weights are different:

		Time(in seconds)		Accuracy (in %)	
Learn(%)	Test(%)	Algo(1)	Algo(2)	(Algo(1)	Algo(2)
90	10	6.33	4.44	100.0	100.0
80	20	3.71	3.78	100.0	100.0
70	30	4.81	6.91	100.0	100.0
60	40	3.55	5.25	99.02	100.0
50	50	3.97	8.11	88.67	91.41

Performance Analysis & Discussion:

When everything is same between them, first algorithm which iterates through the whole dataset at once, takes less time to minimize errors. Because, since it will only update it's weights when a wrong outcome occurs, and since there is only 2 classes, chances of wrong outcome is low. But if there is mistake in second class, weight update and further recalculation is done for that group at first which can lead to error for the other class of data.

When weights are different, the result is not straight forward, rather it is mixed up. At 50% and 60% train split, the accuracy of Algo(2) is higher than that of Algo(1). But most of the cases, Algo(1) beats Algo(2). And in 4 out of 5 times, Algo(1) takes less time than Algo(2) which follows same explanation as first case when everything is same between them.

Lab 3

Title: Backpropagation Neural Network Algorithm

Objectives:

- Solve complex problems like XOR problem that were unsolvable by single layer perceptron algorithm
- Classify multiple classes of data

Theory:

Back propagation is one of the most important achievement in the field of artificial neural network. It solved many problems which were previously insolvable by single layer neural network for example: XOR problem. This algorithm created huge attraction towards this field.

It is more complex than single layer neural network. At first, we present an input to the network. It produces random output. There is an error function that calculates the difference between the networks current output and the current desired output. To learn successfully we want to minimise the error gradually. To achieve this, weight and thresholds of the neuron of the layers are adjusted by the algorithm. Generalised delta rule calculates this error and back propagating the error to the previous layers (hence the name!).

Methodology:

Back-Propagation Neural Network Algorithm

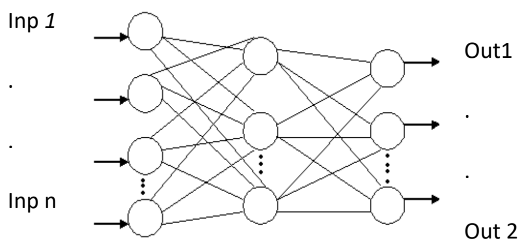


Fig. 1: Three layer neural network

The first step is initializes the weight vectors Wij and W_{jk} and the threshold values for each PE (processing element) with minimum random numbers.

In second step, the network provides the input patterns and the desired respective output patterns.

In third step, the input patterns are connected to the hidden PEs through the weights W_{ij}. In the hidden layer, each PE computed the weighted sum according to the equation,

$$net_{aj} = \sum W_{ij} O_{ai} \qquad (1$$

Where Oai is the input of unit i for pattern number a. The threshold of each PE was then added to its weighted sum to obtain the activation active (j) of that PE i.e.

$$activ_j = net_{aj} + uh_j \qquad (2)$$

Where uh_j is the hidden threshold weights for j^{th} PEs. This activation determined whether the output of the respective PE was either 1 or 0 (fires or not) by using a sigmoid function,

$$O_{aj} = 1/(1+e^{-k_1 * activ_j}) \qquad (3)$$

Where k_1 is called the spread factors, these O_{aj} were then served as the input to the output computation. Signal O_{aj} were then fan out to the output layer according to the relation,

$$net_{ak} = \sum W_{jk} O_{aj} \qquad (4)$$

And the output threshold weight uo_k for k^{th} output PEs was added to it to find out the activation $activo_k$

$$activo_k = net_{ak} + uo_j \qquad (5)$$

The actual output O_{ak} was computed using the same sigmoid function which was

$$O_{ak} = 1/(1+e^{-k_1 * activo_k}) \qquad (6$$

Here another spread factor k_2 has been employed for the output units.

In the second stages, after computing the feed-forward propagation, an error was computed by comparing the output O_{ak} with the respective target t_{ak}, i.e.

$$\delta_{ak} = t_{ak} - O_{ak} \qquad (7)$$

This error was then used to adjust the weights vector W_{jk} using the equation,

$$\Delta W_{jk} = \eta_2 k_2 \delta_{ak} O_{aj} O_{ak} (1 - O_{ak}) \qquad (8)$$

Where $\int^{/}$ ($activo_k$)=k_2 O_{ak} (1- O_{ak}), the derivation of sigmoid function.

The weight vector W_{jk} was then adjusted to

$$W_{jk} = W_{jk} + \Delta W_{jk} \qquad (9)$$

For the threshold weight of the output PE, similar equation was employed,

$$\Delta uo_k = \eta_2 k_2 \delta_{ak} O_{ak} (1 - O_{ak}) \qquad (10)$$

and the new threshold weight equaled as,

$$U_{ok} = U_{ok} + \Delta U_{ok} \qquad (11)$$

In the next step, this error and the adjusted weight vector W_{jk} were feedback to the hidden layer adjust the weight vector W_{ij} and threshold weight uh_j. In this layer, the weight vector W_{ij} was computed by using equation,

$$\Delta W_{ij} = \eta_1 k_1 O_{ai} O_{aj} (1 - O_{aj}) \sum \delta_{ak} W_{jk} \qquad (12)$$

Where $\int^{/}$ ($activh_j$) =k_1 O_{aj} (1- O_{aj}). The weight W_{ij} was then adjusted to

$$W_{ij} = W_{ij} + \Delta W_{ij} \qquad (13)$$

For the threshold weights of the hidden PEs, similar equation was employed,

$$\Delta uh_j = \eta_1 k_1 O_{aj}(1 - O_{aj}) \sum \delta_{ak} W_{jl} \quad (14$$

and the new threshold were calculated

$$uh_j = uh_j + \Delta uh_j \quad (15)$$

Calculating Errors:

After getting the output from the output layer, we calculate the error according to the targeted output in the following error calculating formula,

$$Error_a = 0.5 \sum (t_{ak} - o_{ak})^2 \quad (16$$

Code:

```
#Code for Multi Layer Perceptron
learning
import numpy as np
import matplotlib.pyplot as plt
from numpy import binary_repr
import random

def find_class(num, class_range):
   """returns the class by taking binary
input,
   converting it into decimal and
finding the index of map
   from which th wdecimal is lower."""
   number = ''.join(map(str, num))
   given = int(number, 2)
   for i in range(len(class_range)):
      if given <= class_range[i]:
         return i

def genData(input_line, nb_classes):
   """Generates binary numbers and
class labels. Number of binary
number = 2 ^ input_line"""
   print('Input Lines: ' + str(input_line))
# no of bits in the input, 101 has 3 bits
so 3 input line in perceptron\n",
   number_of_elements =
pow(2,input_line) #total
combination\n",
   nb_output_nodes =
int(np.log2(nb_classes))

   class_difference =
int(number_of_elements /
nb_classes)

   class_range = {}
   j = 0
   for i in range(class_difference-1,
number_of_elements,
class_difference):
      class_range[j] = i
      j = j + 1

   # Making inputs\n",
   inputs = []
   classes = []

   i = 0
   for i in
range(0,number_of_elements, 1):
      tmp =
list(binary_repr(i,input_line))
#params=value, length, this returns
binary patterns for value of length\n",

classes.append(list(binary_repr(find_c
lass(tmp,
class_range),nb_output_nodes)))
      inputs.append(tmp)
      i = i+1
   inputs = np.array(inputs)
   classes = np.array(classes)
   return (inputs.astype(np.float),
classes.astype(np.int)) #convert string
into float and int"
```

```
def split_dataset(inputs, classes, ratio): #split data with train data ratio=ratio\n",
    no = round((len(inputs) * ratio)/2) #0-40 data
    no_test = round(len(inputs)/2) – no #last 40-50 data
    train_x= []
    train_y= []
    test_x= []
    test_y= []
    for i in range(0, no):
        train_x.append(inputs[i])
        train_y.append(classes[i])
    for i in range(no, no+no_test):
        test_x.append(inputs[i])
        test_y.append(classes[i])
    for i in range(round(len(inputs)/2), no+round(len(inputs)/2)):
        train_x.append(inputs[i])
        train_y.append(classes[i])
    for i in range(no+round(len(inputs)/2), len(inputs)):
        test_x.append(inputs[i])
        test_y.append(classes[i])

    return (np.array(train_x), np.array(train_y), np.array(test_x), np.array(test_y))

def sigmoid(k, x):
    return (1/(1+np.exp(-k*x)))

def error(x, y):
    z = sum(list([(i-j)**2 for i,j in zip(x, y)])) #like bitwise operation
    z = 0.5 * z
    return z

def train_perceptron(Oai, Tak, nb_hidden):
    np.random.seed(1)
    nb_input_layer_node = len(Oai[0])
    nb_hidden_layer_node = nb_hidden
    nb_output_layer_node = len(Tak[0])
    nb_input = len(Oai)
    #step 1
    #initializing random weights
    w_ij = np.random.random((nb_hidden_layer_node,nb_input_layer_node))
    w_jk = np.random.random((nb_output_layer_node,nb_hidden_layer_node))
    print("weight ij:",w_ij)
    print("weight jk",w_jk)

    #initializing random threshold values
    uh_j = np.random.random(nb_hidden_layer_node)
    uo_k = np.random.random(nb_output_layer_node)
    print("threshold uhj",uh_j)
    print("threshold uok",uo_k)

    k1= np.random.random()
    k2= np.random.random()
    print ("k1",k1)
    print("k2",k2)

    ita1 = np.random.random()
    ita2 = np.random.random()
    print("ita1",ita1)
    print("ita2",ita2)

    active = []
    pos = 0
    epoch = 0
    count = 0
    while(1):
        print("\nEpoch: "+str(int(epoch/nb_input)))
        print("Input:", Oai[pos])
        net = []
        k = 0
        for m in range(nb_hidden_layer_node):
            net.append(float(np.dot(Oai[pos], w_ij[k].T)))
```

```
        k = k + 1

    active = net + uh_j

    Oaj = []
    for i in
range(nb_hidden_layer_node):
        Oaj.append(sigmoid(k1,
active[i]))
    NETak = []
    k = 0
    for m in
range(nb_output_layer_node):

NETak.append(float(np.dot(Oaj,
w_jk[k].T)))
        k = k + 1
    activek= NETak + uo_k

    Oak = []
    for i in
range(nb_output_layer_node):
        Oak.append(sigmoid(k2,
activek[i]))
    delta = Tak[pos] - Oak
    print("Predicted Output:",Oak)
    print("Actual output", Tak[pos])

    Error = error(Tak[pos], Oak)
    print("Error"+str(Error) + "\n")

    tmpy = list(1-np.array(Oak))
    tmpx=[i*j for i,j in zip(tmpy, Oak)]
    tmpz = np.array([i*j for i,j in
zip(tmpx, delta)])
    del_w_jk = ita2 * k2
    del_w_jk = del_w_jk * tmpz
    del_w_jk_1 = []

    for j in del_w_jk:
        for i in
range(nb_hidden_layer_node):
            del_w_jk_1.append(Oaj[i]*j)

    w_jk = w_jk +
np.reshape(del_w_jk_1,
(nb_output_layer_node,nb_hidden_la
yer_node)) # editing

    tmpx = np.array([i*j for i,j in
zip(Oak, tmpy)])
    tmpz = np.array([i*j for i,j in
zip(tmpx, delta)])
    del_uok = ita2 * k2 *tmpz
    uo_k = uo_k + del_uok

    sum_del_w = []
    m = 0
    for j in delta:
        sum_del_w.append(sum([i*j
for i in w_jk[m]]))
        m = m + 1
    tmpy1 = list(1-np.array(Oaj))
    tmpx1 = np.array([i*j for i,j in
zip(tmpy1, Oaj)])

    tmpz1 = []
    for i in
range(nb_hidden_layer_node):
        tmpz1.append(sum([tmpx1[i]*j
for j in  sum_del_w]))

    del_w_ij = ita1 * k1 *
np.array(tmpz1)

    del_w_ij_1 = []
    for j in del_w_ij:
        for i in
range(nb_input_layer_node):

del_w_ij_1.append(Oai[pos][i] * j)
    #del_w_ij_1 = np.array([i*j for i,j
in zip(Oai[pos], del_w_ij)])

    w_ij = w_ij +
np.reshape(del_w_ij_1,
(nb_hidden_layer_node,nb_input_lay
er_node))

    del_uhj = del_w_ij
    pos = pos + 1
    if(pos == nb_input):
        pos = 0

    epoch = epoch + 1
    if(round(Error, 2) <= 0.10):
```

```
            if(count == nb_input - 1):
                break
            else:
                count = count + 1
        else:
            pos = 0
            count = 0

    return (w_ij, w_jk, uh_j, uo_k, k1,
k2, nb_hidden_layer_node, epoch)

def test_perceptron(Oai, Tak, w_ij,
w_jk, uh_j, uo_k, k1, k2,
nb_hidden_layer_node):
    nb_input = len(Oai)
    nb_output_layer_node =
len(Tak[0])
    e = []
    for pos in range(nb_input):
        print("Input:", Oai[pos])
        net = []
        k = 0
        for m in
range(nb_hidden_layer_node):

net.append(float(np.dot(Oai[pos],
w_ij[k].T)))
            k = k + 1

        active = net + uh_j

        Oaj = []
        for i in
range(nb_hidden_layer_node):
            Oaj.append(sigmoid(k1,
active[i]))

        NETak = []
        k = 0
        for m in
range(nb_output_layer_node):

NETak.append(float(np.dot(Oaj,
w_jk[k].T)))
            k = k + 1

        activek= NETak + uo_k
        Oak = []
        for i in
range(nb_output_layer_node):
            Oak.append(sigmoid(k2,
activek[i]))

        print("Predicted Output:",Oak)
        print("Actual output", Tak[pos])
        Error = error(Tak[pos], Oak)
        print("Error"+str(Error) + "\n")
        e.append(Error)
    return e

if __name__=="__main__":
    nb_of_line = 5
    nb_classes = 4
    nb_hidden = 5
    (inputs, classes) =
genData(nb_of_line, nb_classes)

    train_ratio = 0.8
    (train_x, train_y, test_x, test_y) =
split_dataset(inputs, classes,
train_ratio)
    #step 2
    (w_ij, w_jk, uh_j, uo_k, k1, k2,
nb_hidden_layer_node) =
train_perceptron(train_x, train_y,
nb_hidden)
    print("Testing\n")
    e = test_perceptron(test_x, test_y,
w_ij, w_jk, uh_j, uo_k, k1, k2,
nb_hidden_layer_node)
    e = np.array(e)
    print('Average Error: ', np.mean(e))
    print('Standard deviation', np.std(e))
```

Input:

Number of Input Line = 5
Number of classes = 4
Number of hidden layer node = 5
Train ratio = 0.8
k1 0.2804439920644052
k2 0.7892793284514885
ita1 0.10322600657764203
ita2 0.44789352617590517

Output:

weight ij: [
[4.17022005e-01 7.20324493e-01 1.14374817e-04 3.02332573e-01 1.46755891e-01]
[9.23385948e-02 1.86260211e-01 3.45560727e-01 3.96767474e-01 5.38816734e-01]
[4.19194514e-01 6.85219500e-01 2.04452250e-01 8.78117436e-01 2.73875932e-02]
[6.70467510e-01 4.17304802e-01 5.58689828e-01 1.40386939e-01 1.98101489e-01]
[8.00744569e-01 9.68261576e-01 3.13424178e-01 6.92322616e-01 8.76389152e-01]
]
weight jk [
[0.89460666 0.08504421 0.03905478 0.16983042 0.8781425]
[0.09834683 0.42110763 0.95788953 0.53316528 0.69187711]
]
threshold uhj [0.31551563 0.68650093 0.83462567 0.01828828 0.75014431]
threshold uok [0.98886109 0.74816565]
k1 0.2804439920644052
k2 0.7892793284514885
ita1 0.10322600657764203
ita2 0.44789352617590517
Testing

Input: [0. 1. 1. 0. 1.]
Predicted Output: [0.033543612782787141, 0.98915139655846018]
Actual output [0 1]
Error0.000621433077577

Input: [0. 1. 1. 1. 0.]
Predicted Output: [0.033559540507953262, 0.98912195272445091]
Actual output [0 1]
Error0.000622287335817

Input: [0. 1. 1. 1. 1.]
Predicted Output: [0.033476912514069979, 0.98926900432783749]

Actual output [0 1]
Error0.000617928969795

Input: [1. 1. 1. 0. 1.]
Predicted Output: [0.75182543047209016, 0.85861462071103589]
Actual output [1 1]
Error0.0407902212185

Input: [1. 1. 1. 1. 0.]
Predicted Output: [0.75644429509802502, 0.86054559017869992]
Actual output [1 1]
Error0.0393834569045

Input: [1. 1. 1. 1. 1.]
Predicted Output: [0.72016826471599615, 0.86711838123123997]
Actual output [1 1]
Error0.0479816623393

Average Error: 0.0216694983076
Standard 0.0212167035652
Steps: 3778

Performance Analysis & Discussion:

Back propagation neural net takes input and calculates an output. If the output is not the same as desired or the error is not less than 10% then the algorithm continues with the same input. If less than 10% error is encountered, the algorithm moves on with the next input line. At last, the weights of the hidden layer and output layer along with the thresholds value of the nodes of hidden and output layers are saved and used during testing phase

Lab 4

Title: Kohonen Self-Organising Network

Objectives:

- Study of an unsupervised neural network
- Map input patterns into output grid

Theory:

Kohonen self-organising network is an unsupervised neural network algorithm that finds the common features among input patterns. The learning algorithm organizes the nodes in the grid into local neighbourhood that acts as a feature classifiers on the input data. The topographic map is autonomously organized by a cyclic process by comparing input patterns to vectors stored at each node. No training response is specified for any input.

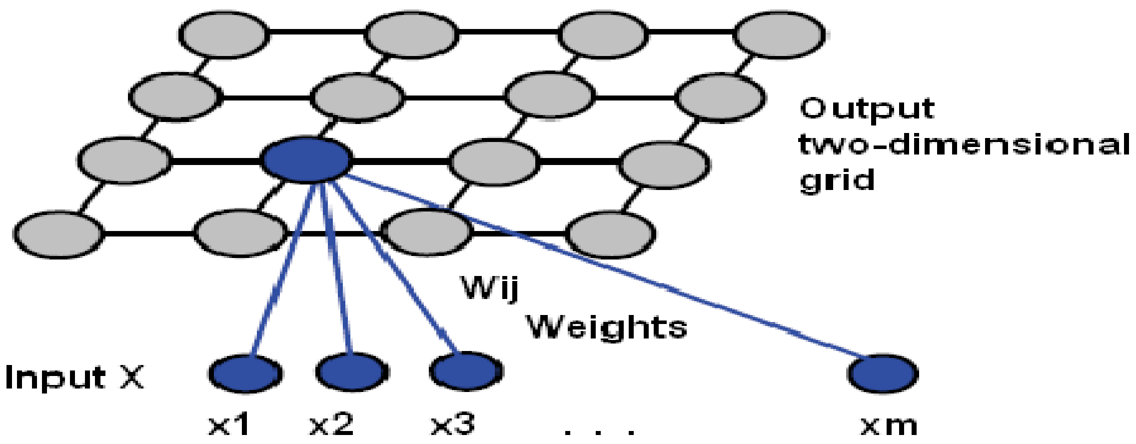


Fig 1: Kohonen Network

Methodology:

1. Initialize network
   Define $W_{ij}(t)$ (0<= i <= n-1) to be the weight from input I to node j at time t. Inintialize the network from the n inputs to the nodes to small randor values. Set the initial radius of the neighbourhood around node j, Nj (0) to be large
2. Present Input
   Present input $x_0(t)$, $x_1(t)$, $x_2(t)$..... $x_{n-1}(t)$ where $x_i(t)$ is the input to node I at time t.
3. Calculate distances
   Calculate the distance d_j between the input and each output node j given by
   $$d_j = \sum_{i=0}^{n-1} (x_i(t) - w_{ij}(t))^2$$
4. Select minimum distance

Designate the output node with minimum d_j to be j*

5. Update weights
   Update weights for node j* and its neighbors defined by the neighbourhood soze $N_{j*}(t)$. New weights are
   $$W_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t)-w_{ij}(t))$$
   For j in $N_{j*}(t)$, 0 <=i <= n-1

   The term $\eta(t)$ is the gain term. Its value is between 0 and 1 inclusive.

6. Repeat by going to 2.

Code:

```
#Code for kohonen neural network
import numpy as np
import matplotlib.pyplot as plt
from numpy import binary_repr
import random
np.random.seed(100)

def genData(input_line):
   print('Input Lines: ' + str(input_line))
# no of bits in the input, 101 has 3 bits
so 3 input line in perceptron\n",
   number_of_elements =
pow(2,input_line) #total
combination\n",


   # Making inputs\n",
   inputs = []

   i = 0
   for i in
range(0,number_of_elements, 1):
      tmp =
list(binary_repr(i,input_line))
#params=value, length, this returns
binary patterns for value of length\n",
      inputs.append(tmp)
      i = i+1
   inputs = np.array(inputs)
   return (inputs.astype(np.float))
#convert string into float and int"
def split_dataset(inputs, ratio): #split
data with train data ratio=ratio\n",
   no = round((len(inputs) * ratio)/2)
   no_test = round(len(inputs)/2) - no
   train_x= []
   test_x= []
   for i in range(0, no):
      train_x.append(inputs[i])
   for i in range(no, no+no_test):
      test_x.append(inputs[i])
   for i in range(round(len(inputs)/2),
no+round(len(inputs)/2)):
      train_x.append(inputs[i])
   for i in
range(no+round(len(inputs)/2),
len(inputs)):
      test_x.append(inputs[i])

   return (np.array(train_x),
np.array(test_x))

def calculate_distance(x, w):
   return sum((x-w)**2)

def train_kohonen(x, output_class):
   nb_input = len(x) #total input
   nb_input_layer_node = len(x[0])
#number of input layer node
   W =
np.random.random((output_class,
nb_input_layer_node)) #weights
   final_W = []
   distance = [[0, 0] for i in
range(output_class)]
   N = 3 #initial radius
   ita = np.random.random() #gain
term
```

```
    i = 0
    flag = False
    k = 0
    while True:
        if i == nb_input:
            break
        for j in range(output_class):
            distance[j][0] =
calculate_distance(x[i], W[j])
            distance[j][1] = j
        dist =
sorted(distance,key=lambda l:l[0])

        #update weights
        for j in range(N):
            W[dist[j][1]] = W[dist[j][1]] +
ita*(x[i]-W[dist[j][1]])#dist[j][1] will give
index with lowest distance
        if flag == True:
            i = i + 1 #next sample
            flag = False #resetting flag
            N = 3 #reset neighbor
            final_W.append([W[dist[0][1]],
dist[0][1]])
            continue

        N = round(N - ita * N)
        if N == 1:
            flag = True
    return (final_W, output_class, ita)

def test_kohonen(x, W, output_class,
ita):
    nb_input = len(x)
    distance = [[0, 0] for i in
range(output_class)]

    for i in range(nb_input):
        for j in range(output_class):
                distance[j][0] =
calculate_distance(x[i], W[j][0])
                distance[j][1] = j
        dist =
sorted(distance,key=lambda l:l[0])
        print('Pattern:', x[i], 'Predicted:',
dist[0][1])

if __name__=="__main__":
    nb_of_line = 5
    nb_hidden = 5
    inputs = genData(nb_of_line)

    train_ratio = 0.8
    (train_x, test_x) =
split_dataset(inputs,train_ratio)

    (final_W, output_class, ita) =
train_kohonen(train_x, nb_hidden)
    k = 0
    for i, j in final_W:
        print('Pattern:',train_x[k],
'Predicted:', j)
        k = k + 1
    print('\nTesting\n')
    test_kohonen(test_x, final_W,
output_class, ita)
```

Input:
No of input line = 5
No of output grid node = 5
Train ratio = 0.8

Output:

```
Pattern: [ 0.  0.  0.  0.  0.] Predicted: 3
Pattern: [ 0.  0.  0.  0.  1.] Predicted: 1
Pattern: [ 0.  0.  0.  1.  0.] Predicted: 0
Pattern: [ 0.  0.  0.  1.  1.] Predicted: 1
Pattern: [ 0.  0.  1.  0.  0.] Predicted: 2
Pattern: [ 0.  0.  1.  0.  1.] Predicted: 2
Pattern: [ 0.  0.  1.  1.  0.] Predicted: 0
Pattern: [ 0.  0.  1.  1.  1.] Predicted: 4
Pattern: [ 0.  1.  0.  0.  0.] Predicted: 3
Pattern: [ 0.  1.  0.  0.  1.] Predicted: 1
Pattern: [ 0.  1.  0.  1.  0.] Predicted: 0
Pattern: [ 0.  1.  0.  1.  1.] Predicted: 1
Pattern: [ 0.  1.  1.  0.  0.] Predicted: 0
Pattern: [ 1.  0.  0.  0.  0.] Predicted: 2
Pattern: [ 1.  0.  0.  0.  1.] Predicted: 2
Pattern: [ 1.  0.  0.  1.  0.] Predicted: 3
Pattern: [ 1.  0.  0.  1.  1.] Predicted: 2
Pattern: [ 1.  0.  1.  0.  0.] Predicted: 3
Pattern: [ 1.  0.  1.  0.  1.] Predicted: 0
Pattern: [ 1.  0.  1.  1.  0.] Predicted: 2
Pattern: [ 1.  0.  1.  1.  1.] Predicted: 0
Pattern: [ 1.  1.  0.  0.  0.] Predicted: 1
Pattern: [ 1.  1.  0.  0.  1.] Predicted: 1
```

Pattern: [1. 1. 0. 1. 0.] Predicted: 2
Pattern: [1. 1. 0. 1. 1.] Predicted: 1
Pattern: [1. 1. 1. 0. 0.] Predicted: 2

Testing

Pattern: [0. 1. 1. 0. 1.] Predicted: 0
Pattern: [0. 1. 1. 1. 0.] Predicted: 4
Pattern: [0. 1. 1. 1. 1.] Predicted: 1
Pattern: [1. 1. 1. 0. 1.] Predicted: 0
Pattern: [1. 1. 1. 1. 0.] Predicted: 4
Pattern: [1. 1. 1. 1. 1.] Predicted: 1

Performance Analysis & Discussion:

Kohonen takes an input pattern, calculates the distance and takes the node with shortest distance as the response of that input pattern. Kohonen then save the weight vectors for that node along with the node number. These are used during testing phase. That is what we have seen in the output of the network.

Lab 5

Title: Hopfield Network

Objectives:

- Converting unknown patterns into a learned pattern
- Recovering known pattern from noisy pattern.

Theory:

Hopfield Network is a fully connected, symmetrically weighted network. It is a fully connected mesh network. Every node is connected to all the other nodes except itself.

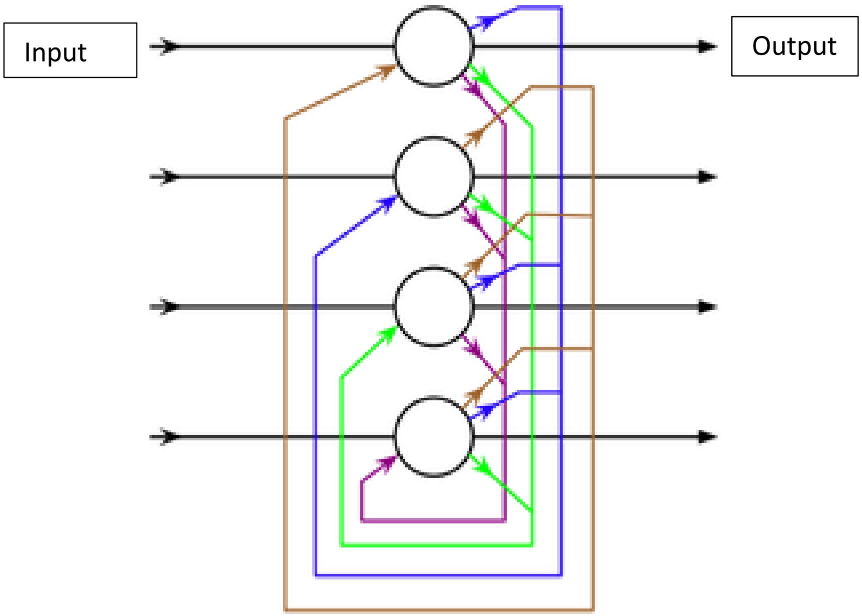


Fig 1: Hopfield Network

It uses bipolar inputs generally, but can work with binary inputs too. The input and output nodes are the same. It calculates the required weights first and then input is applied. So, there is no separate training phase for this network.

Methodology:

1. Assign connection weights

$$w_{ij} = \begin{cases} \sum_{s=0}^{M-1} x_i(s)x_j(s), & i \mathrel{!}= j \\ 0, \ i = j, 0 \leq i,j \leq M-1 \end{cases}$$

Where w_{ij} is the connection weight between node i and node j, and $x_i(s)$ is the element of node I of the exemplar pattern for class s. The threshold value of the nodes are zero.

2. Initialize with unknown pattern

   $U_i(0) = x_i \quad 0 <= i <= N-1$

3. Iterate until convergence

   $$U_i(t+1) = f_h\left[\sum_{j=0}^{N-1} w_{ij}u_j(t)\right] \quad 0 <= j <= N-1$$

The function f_h is a hard limiting non linearity, the step function. Repeat the iteration until the outputs from the nodes remain unchanged.

Code:

```
#Code for Hopfield neural network
import numpy as np
import matplotlib.pyplot as plt
from numpy import binary_repr
import random
from sklearn.model_selection import
train_test_split
import pandas as pd
np.random.seed(100)

def multiply(x, i, j):
    ans = 0
    for k in range(len(x)):
        ans += x[k][i] * x[k][j]
    return ans

def hard_limiting_threshold(x):
    #print(x)
    if x <= 0:
        return -1.0
    else:
        return 1.0

def n_multiply(w, miu, nb_node):
    ans = []
    for i in range(nb_node):

ans.append(hard_limiting_threshold(
np.dot(np.array(w[i]).T,
np.array(miu))))
    return ans

def match(a, b):
    for i in range(len(a)):
        if round(a[i]) != round(b[i]):
            return False
    return True

def hopfield(x):
    nb_node = len(x[0])
    w = np.zeros((nb_node, nb_node))
    for i in range(nb_node):
        for j in range(nb_node):
            if i is not j:
                w[i][j] = multiply(x, i, j)
    print(w)
    return w

def test_hopfield(x, w):
    nb_node = len(x[0])
    for i in range(len(x)):
        print('\nFor input: ', x[i])
        miu = x[i]
        while True:
            tmp = miu
            miu = n_multiply(w, miu,
nb_node)
            if match(miu , tmp):
                print('Predicted',miu)
                break

if __name__=="__main__":
    train = pd.read_csv('Train-
Data.csv').values
    test = pd.read_csv('Train-
Data.csv').values

    w = hopfield(train)
    test_hopfield(test, w)
```

Input:
Two csv file containing 60 bits input patterns and test patterns.

Output:

Weights:
[[0. 9. 5. ..., 1. 3. 3.]
 [9. 0. 5. ..., 1. 3. 3.]
 [5. 5. 0. ..., -3. 3. 3.]
 ...,
 [1. 1. -3. ..., 0. -1. -1.]
 [3. 3. 3. ..., -1. 0. -3.]
 [3. 3. 3. ..., -1. -3. 0.]]

For input: [-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 -1 1 1 -1]
Predicted [-1.0, 1.0, -1.0, -1.0]
For input: [-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 1 -1 1 -1]
Predicted [-1.0, -1.0, 1.0, -1.0, -1.0, -1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, -1.0, -1.0, 1.0, -1.0, -1.0, 1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, 1.0, 1.0, -1.0, -1.0, -1.0, 1.0, -1.0, -1.0, 1.0, 1.0, -1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, -1.0, 1.0, 1.0]

Performance Analysis & Discussion:
Hopfield network calculates weight and then take unknown pattern and tries to make it as similar as a learned input pattern. It iterates until convergence is reached.