In the name of Allah, the Beneficent, the Merciful.

Assignment No.3

Artificial Neural Networks

Submitted To: Dr. Nasser Mozayani

Submitted by: Muhammad Hasnain Khan

Student I'd: 401722316

Q. No.1:

$\alpha = 0.5$, $P_1 = [1 \quad -1]^T$, $P_2 = [1 \quad 1]^T$, $P_3 = [-1 \quad -1]^T$

$$w = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{2} \end{bmatrix}$$

- **For input Vector P_1:**

$$d_1 = (\sqrt{2} - 1)^2 + (0 + 1)^2 = 2.59$$

$$d_2 = (0 - 1)^2 + (\sqrt{2} + 1)^2 = 5.41$$

The input vector is near to the node 1 of the weights. So, its weights will be updated with the following equation:

$$w_j(new) = w_j(old) + \alpha(x_j - w_j(old))$$

$$w_1 = [\sqrt{2} \quad 0] + (0.5)([1 \quad -1] - [\sqrt{2} \quad 0]) = [-0.795 \quad -0.5]$$

So, the updated weights matrix will be as follows:

$$w_1 = \begin{bmatrix} -0.795 & 0 \\ -0.500 & \sqrt{2} \end{bmatrix}$$

- **For input Vector P_2:**

$$d_1 = (-0.795 - 1)^2 + (-0.5 - 1)^2 = 5.45$$

$$d_2 = (0 - 1)^2 + (\sqrt{2} - 1)^2 = 2.59$$

The input vector is near to the node 2 of the weights. So, its weights will be updated with the following equation:

$$w_j(new) = w_j(old) + \alpha(x_j - w_j(old))$$

$$w_2 = [0 \quad \sqrt{2}] + (0.5)([1 \quad 1] - [0 \quad \sqrt{2}]) = [0.5 \quad 1.2]$$

So, the updated weights matrix will be as follows:

$$w_2 = \begin{bmatrix} -0.795 & 0.5 \\ -0.500 & 1.2 \end{bmatrix}$$

- **For input Vector P$_3$:**

$$d_1 = (-0.795 + 1)^2 + (-0.5 + 1)^2 = 0.29$$

$$d_2 = (0.5 + 1)^2 + (1.2 + 1)^2 = 7.09$$

The input vector is near to the node 1 of the weights. So, its weights will be updated with the following equation:

$$w_j(new) = w_j(old) + \alpha(x_j - w_j(old))$$

$$w_3 = [-0.795 \quad -0.5] + (0.5)([-1 \quad -1] - [-0.795 \quad -0.5]) = [-0.89 \quad -0.75]$$

So, the updated weights matrix will be as follows:

$$w_3 = \begin{bmatrix} -0.89 & 0.5 \\ -0.75 & 1.2 \end{bmatrix}$$

By updating the learning rate α, we can iterate the process.

Ref: https://bioinformatics.cs.vt.edu/~easychair/SOM/Fausett_FundamentalsofNeuralNetworks.pdf
https://cse.engineering.nyu.edu/~mleung/CS6673/s09/SOM.pdf

Q. No.2 (a):

X1	X2	Φ_1	Φ_2	Output
0	0	1	0.1353	0
1	0	0.3678	0.3678	1
0	1	0.3678	0.3678	1
1	1	0.1353	1	1

$$\mu_1 = (0,0),\ \ \mu_2 = (1,1)$$

First calculate the Φ_1 and Φ_2 for the given inputs by following formulas:

$$\Phi_1(\mathrm{x}) \ = \ e^{-(||x-\mu_1||^2)}$$

$$\Phi_2(\mathrm{x}) \ = \ e^{-(||x-\mu_2||^2)}$$

X = (0,0)

$$\Phi_1(\mathrm{x}) \ = \ e^{-((0-0)^2+(0-0)^2)} = e^{-(0)} = 1$$

$$\Phi_2(\mathrm{x}) \ = \ e^{-((0-1)^2+(0-1)^2)} = e^{-(2)} = 0.1353$$

X = (1,0)

$$\Phi_1(\mathrm{x}) \ = \ e^{-((1-0)^2+(0-0)^2)} = e^{-(1)} = 0.3678$$

$$\Phi_2(\mathrm{x}) \ = \ e^{-((1-1)^2+(0-1)^2)} = e^{-(1)} = 0.3678$$

X = (0,1)

$$\Phi_1(\mathrm{x}) \ = \ e^{-((0-0)^2+(1-0)^2)} = e^{-(1)} = 0.3678$$

$$\Phi_2(\mathrm{x}) \ = \ e^{-((0-1)^2+(1-1)^2)} = e^{-(1)} = 0.3678$$

X = (1,1)

$$\Phi_1(\mathrm{x}) \ = \ e^{-((1-0)^2+(1-0)^2)} = e^{-(2)} = 0.1353$$

$$\Phi_2(\mathrm{x}) \ = \ e^{-((1-1)^2+(1-1)^2)} = e^{-(0)} = 1$$

Calculate the weights by the following formula:

$$output = w_1\Phi_1(\mathrm{x}) + w_2\Phi_2(\mathrm{x}) - b$$

From the results of Φ_1 and Φ_2 we get the following equations to calculate weights:

$$0 = \mathrm{w}_1 + 0.1353\mathrm{w}_2 - \mathrm{b}$$

$$1 = 0.3678\mathrm{w}_1 + 0.3678\mathrm{w}_2 - \mathrm{b}$$

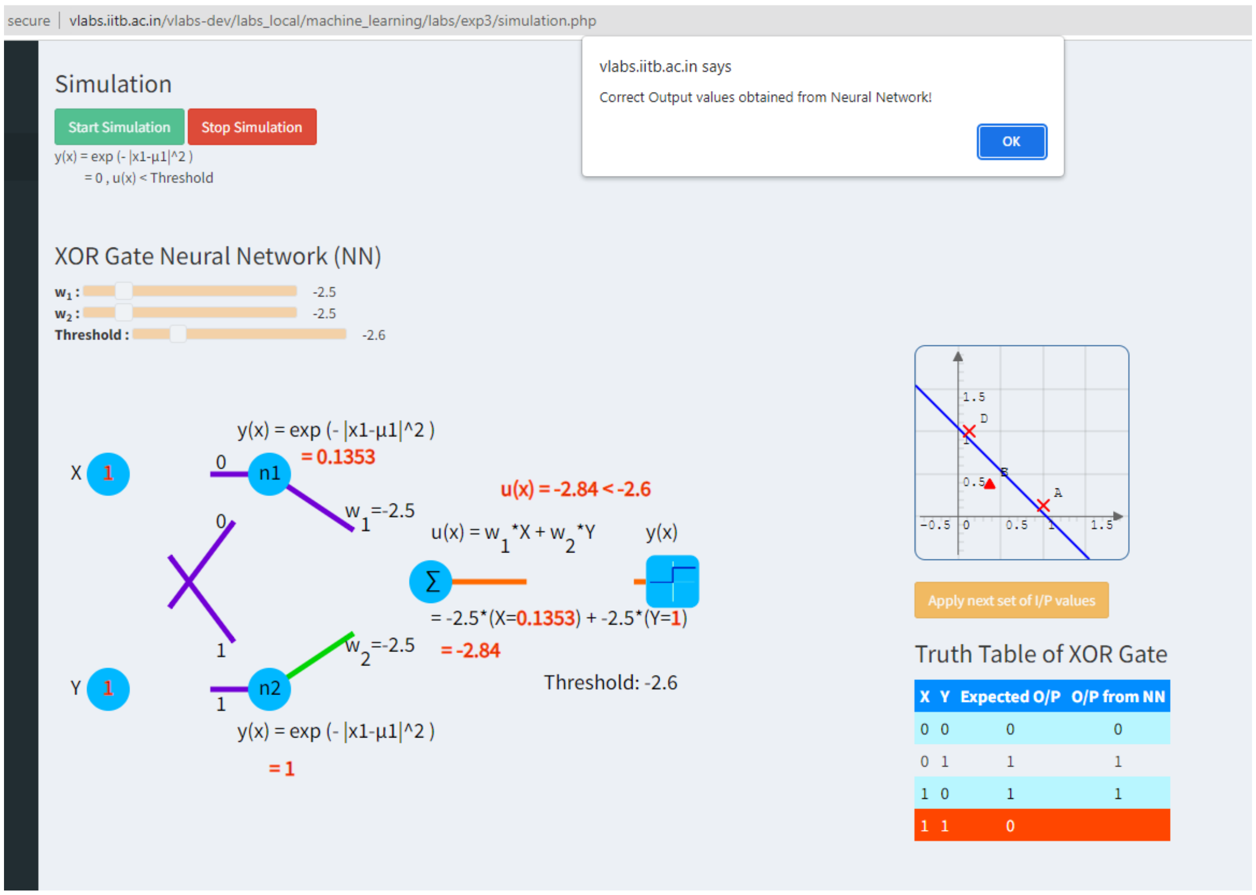
$$1 = 0.3678\mathrm{w}_1 + 0.3678\mathrm{w}_2 - \mathrm{b}$$

$$0 = 0.1353\mathrm{w}_1 + \mathrm{w}_2 - \mathrm{b}$$

$$\mathrm{w}_1 = \mathrm{w}_2 = -2.5018\,, \qquad bias\ or\ b = -2.8404$$

Q. No.2 (b):

The calculated weights are implemented in the simulation at the given website and obtained the correct output values, the results are as follows:



secure | vlabs.iitb.ac.in/vlabs-dev/labs_local/machine_learning/labs/exp3/simulation.php

vlabs.iitb.ac.in says

Correct Output values obtained from Neural Network!

OK

Simulation

Start Simulation | Stop Simulation

y(x) = exp (- |x1-µ1|^2)
= 0 , u(x) < Threshold

XOR Gate Neural Network (NN)

w_1 : -2.5
w_2 : -2.5
Threshold : -2.6

y(x) = exp (- |x1-µ1|^2)
= 0.1353

u(x) = -2.84 < -2.6

u(x) = w_1*X + w_2*Y

= -2.5*(X=0.1353) + -2.5*(Y=1)
= -2.84

Threshold: -2.6

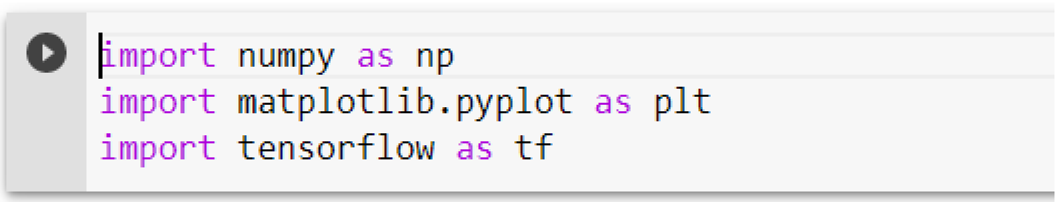
y(x) = exp (- |x1-µ1|^2)
= 1

Truth Table of XOR Gate

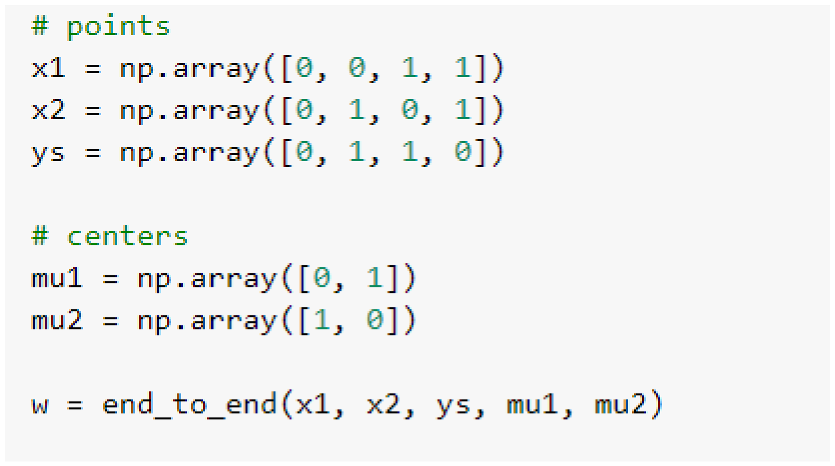
X	Y	Expected O/P	O/P from NN
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	

Q. No.2 (c):

1. To understand the Radial Basis Function Network, which is implemented in the RBF.ipynb we will start from importing libraries and understanding the input data. Which is given bellow:



```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
```



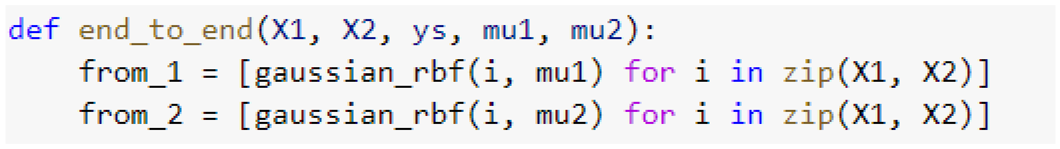
```
# points
x1 = np.array([0, 0, 1, 1])
x2 = np.array([0, 1, 0, 1])
ys = np.array([0, 1, 1, 0])

# centers
mu1 = np.array([0, 1])
mu2 = np.array([1, 0])

w = end_to_end(x1, x2, ys, mu1, mu2)
```

a. Here x1, x2 are input vectors and ys is the expected output vector.
b. mu1 and mu2 are the coordinates of the centers
c. w performs end_to_end function for calculating the weights according to the inputs (x1, x2) and centers (mu1, mu2)

2. end_to_end function:

This function has three parts:

First Part:



```
def end_to_end(X1, X2, ys, mu1, mu2):
    from_1 = [gaussian_rbf(i, mu1) for i in zip(X1, X2)]
    from_2 = [gaussian_rbf(i, mu2) for i in zip(X1, X2)]
```

a) This function takes the inputs, output and centers
b) from_1 calculates the RBF activation function using Gaussian function for center 1 (mu1)
c) from_2 calculates the RBF activation function using Gaussian function for center 2 (mu2)
d) The Gaussian Function (gaussian_rbf (x, landmark, gamma=1)) is defined for the Radial Basis Function, where x takes input values from x1 and x2, landmark takes the values of mu1 and mu2 while gamma is being considered as 1.

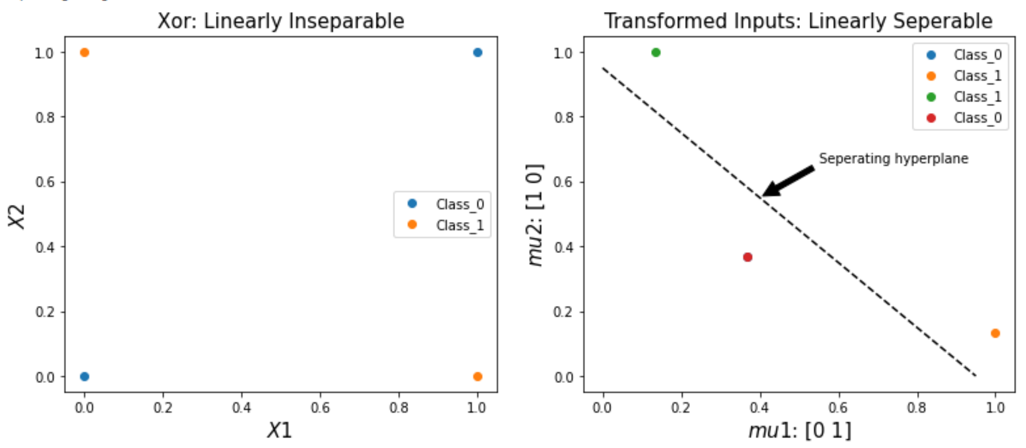
```python
def gaussian_rbf(x, landmark, gamma=1):
    return np.exp(-gamma * np.linalg.norm(x - landmark)**2)
```

Second Part:

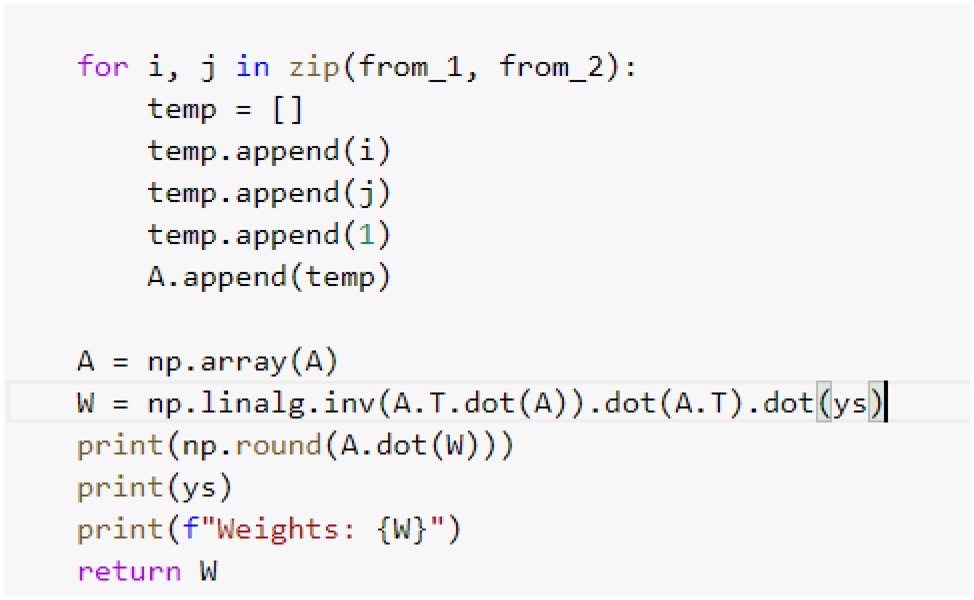
Second part plots the inseparable and separable classifications before and after applying the Gaussian function or before and after transforming the inputs.

```python
plt.figure(figsize=(13, 5))
plt.subplot(1, 2, 1)
plt.scatter((x1[0], x1[3]), (x2[0], x2[3]), label="Class_0")
plt.scatter((x1[1], x1[2]), (x2[1], x2[2]), label="Class_1")
plt.xlabel("$X1$", fontsize=15)
plt.ylabel("$X2$", fontsize=15)
plt.title("Xor: Linearly Inseparable", fontsize=15)
plt.legend()

plt.subplot(1, 2, 2)
plt.scatter(from_1[0], from_2[0], label="Class_0")
plt.scatter(from_1[1], from_2[1], label="Class_1")
plt.scatter(from_1[2], from_2[2], label="Class_1")
plt.scatter(from_1[3], from_2[3], label="Class_0")
plt.plot([0, 0.95], [0.95, 0], "k--")
plt.annotate("Seperating hyperplane", xy=(0.4, 0.55), xytext=(0.55, 0.66),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.xlabel(f"$mu1$: {(mu1)}", fontsize=15)
plt.ylabel(f"$mu2$: {(mu2)}", fontsize=15)
plt.title("Transformed Inputs: Linearly Seperable", fontsize=15)
plt.legend()
```

Its output is as follows:

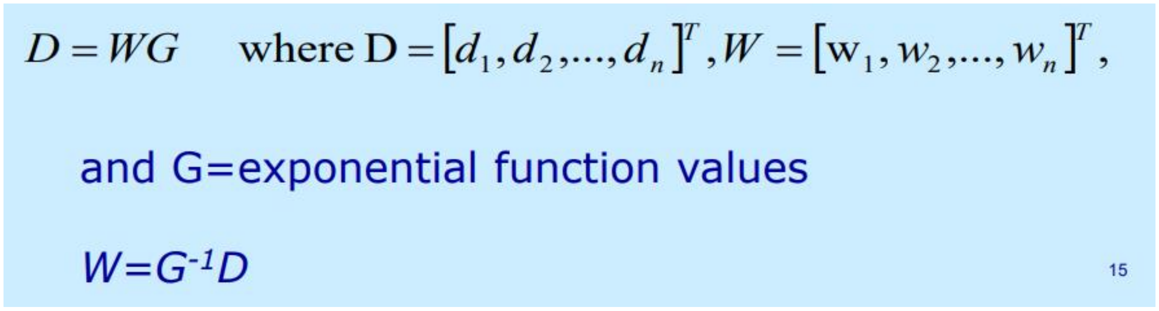


Third Part:



```
for i, j in zip(from_1, from_2):
    temp = []
    temp.append(i)
    temp.append(j)
    temp.append(1)
    A.append(temp)

A = np.array(A)
W = np.linalg.inv(A.T.dot(A)).dot(A.T).dot(ys)
print(np.round(A.dot(W)))
print(ys)
print(f"Weights: {W}")
return W
```

a) A for loop is initiated to append the transformed inputs or exponential functions values in a vector A
b) Then W calculates the weights of the inputs by applying the following formula for the weights calculations (from the lecture slides)



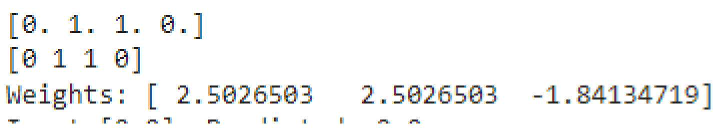
$D = WG$ where $D = [d_1, d_2, ..., d_n]^T, W = [w_1, w_2, ..., w_n]^T$,

and G=exponential function values

$W = G^{-1}D$

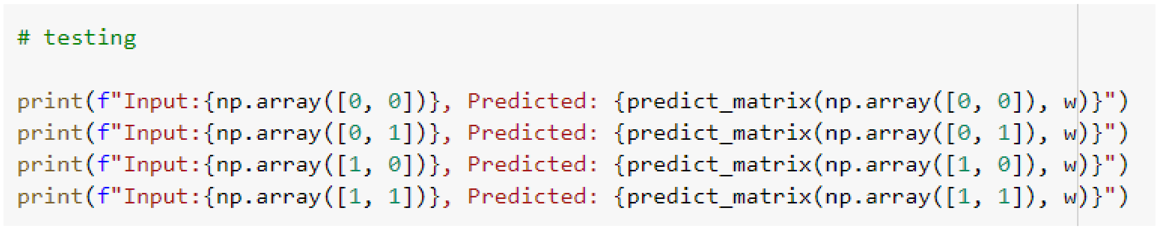
15

c) Then the ys (the expected output) is checked and printed with the current output of the transformed inputs by taking dot product of transformed inputs and their weights.
d) At last the weights are printed and these outputs are shows as follows:



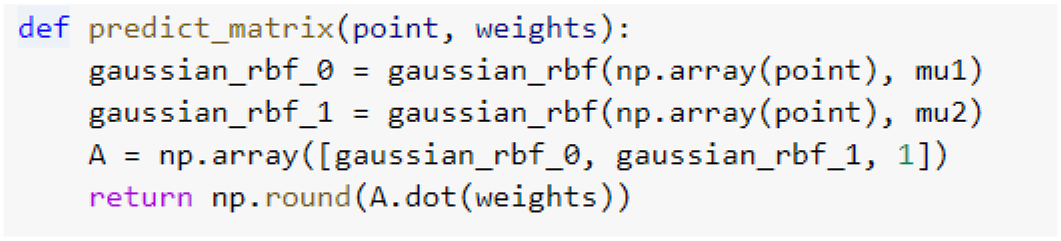
```
[0. 1. 1. 0.]
[0 1 1 0]
Weights: [ 2.5026503   2.5026503  -1.84134719]
```

3. At the end is the testing of the RBF network:



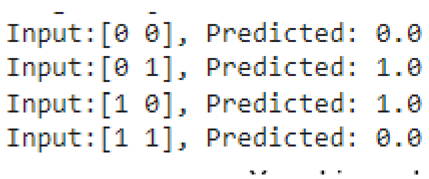
```
# testing

print(f"Input:{np.array([0, 0])}, Predicted: {predict_matrix(np.array([0, 0]), w)}")
print(f"Input:{np.array([0, 1])}, Predicted: {predict_matrix(np.array([0, 1]), w)}")
print(f"Input:{np.array([1, 0])}, Predicted: {predict_matrix(np.array([1, 0]), w)}")
print(f"Input:{np.array([1, 1])}, Predicted: {predict_matrix(np.array([1, 1]), w)}")
```

a) In testing every input is printed with its predicted or expected output
b) Predicted is being obtained by applying predict_matrix function
c) predict_matrix function is defined as:



```
def predict_matrix(point, weights):
    gaussian_rbf_0 = gaussian_rbf(np.array(point), mu1)
    gaussian_rbf_1 = gaussian_rbf(np.array(point), mu2)
    A = np.array([gaussian_rbf_0, gaussian_rbf_1, 1])
    return np.round(A.dot(weights))
```

i) this function takes points and weights
ii) Gaussian function is calculated for each point by applying gaussian_rbf on each point for mu1 and mu2
iii) Then A, which is a vector of each transformed point with respect to mu1 and mu2.
iv) At the end the dot product of A and the weighs is calculated to obtain the predicted or expected outputs which shows that the weights which obtained are correct.
v) The output for testing is as follows:



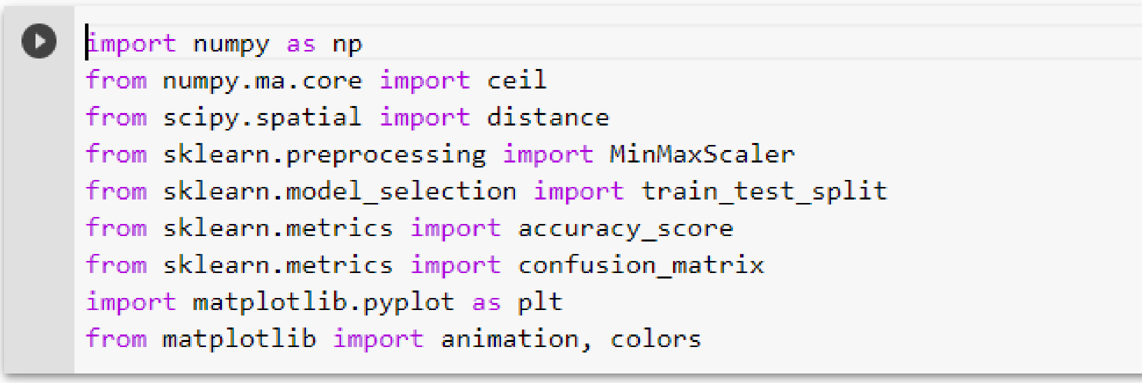
```
Input:[0 0], Predicted: 0.0
Input:[0 1], Predicted: 1.0
Input:[1 0], Predicted: 1.0
Input:[1 1], Predicted: 0.0
```

Q. No.3:

Explanation of SOM model for Banknote Authentication:

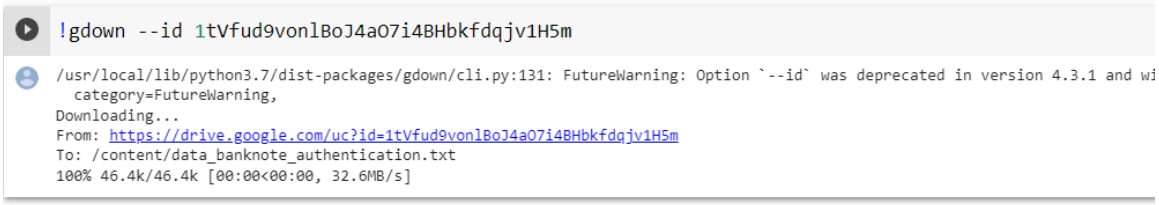
As from the given code for this model has the following steps:

1) Importing Libraries for the model

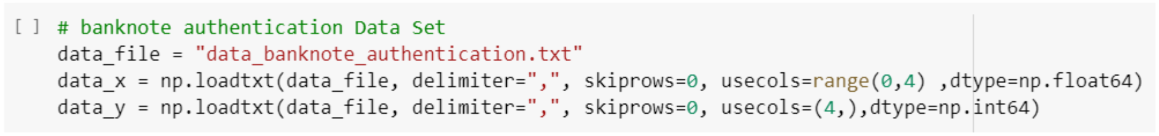


```
import numpy as np
from numpy.ma.core import ceil
from scipy.spatial import distance
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from matplotlib import animation, colors
```

2) Downloading and Loading Dataset



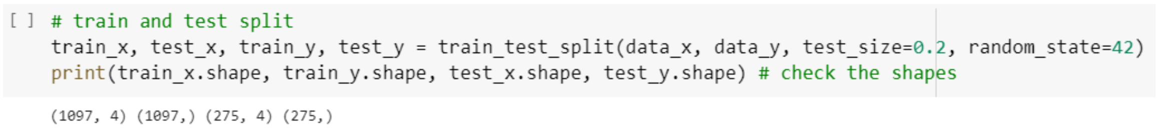
```
!gdown --id 1tVfud9vonlBoJ4a07i4BHbkfdqjv1H5m

/usr/local/lib/python3.7/dist-packages/gdown/cli.py:131: FutureWarning: Option `--id` was deprecated in version 4.3.1 and wi
  category=FutureWarning,
Downloading...
From: https://drive.google.com/uc?id=1tVfud9vonlBoJ4a07i4BHbkfdqjv1H5m
To: /content/data_banknote_authentication.txt
100% 46.4k/46.4k [00:00<00:00, 32.6MB/s]
```



```
[ ] # banknote authentication Data Set
    data_file = "data_banknote_authentication.txt"
    data_x = np.loadtxt(data_file, delimiter=",", skiprows=0, usecols=range(0,4) ,dtype=np.float64)
    data_y = np.loadtxt(data_file, delimiter=",", skiprows=0, usecols=(4,),dtype=np.int64)
```

The CSV file is downloaded from the Google drive and stored in the directory.
data_x uses the first four columns as x, and data_y uses the last column as y

3) Training and Testing Data Split



```
[ ] # train and test split
    train_x, test_x, train_y, test_y = train_test_split(data_x, data_y, test_size=0.2, random_state=42)
    print(train_x.shape, train_y.shape, test_x.shape, test_y.shape) # check the shapes

    (1097, 4) (1097,) (275, 4) (275,)
```

The data is split for training and testing with test_size = 0.2 and the output shows the trained data's shape.

4) Required functions for the model

i) min_scaler(data)

```
# Data Normalisation
def minmax_scaler(data):
  scaler = MinMaxScaler()
  scaled = scaler.fit_transform(data)
  return scaled
```

This function is used to normalize the input data between 0 and 1.

ii) e_distance(x,y)

```
# Euclidean distance
def e_distance(x,y):
  return distance.euclidean(x,y)
```

This function calculates the Euclidean distance between the two points. Euclidean distance is used to search for the winning neuron.

iii) m_distance(x,y)

```
# Manhattan distance
def m_distance(x,y):
  return distance.cityblock(x,y)
```

This function calculates the Manhattan distance between two points. the Manhattan distance is used to limit the neighborhood range.

iv) winning_neuron()

```
# Best Matching Unit search
def winning_neuron(data, t, som, num_rows, num_cols):
  winner = [0,0]
  shortest_distance = np.sqrt(data.shape[1]) # initialise with max distance
  input_data = data[t]
  for row in range(num_rows):
    for col in range(num_cols):
      distance = e_distance(som[row][col], data[t])
      if distance < shortest_distance:
        shortest_distance = distance
        winner = [row,col]
  return winner
```

This function searches the best matching unit (BMU) for the sample data t. The distance between the input signal and every neuron in the map layer is calculated and the row and column index of the grid of the neuron with the shortest distance is returned.

v) decay()

```
# Learning rate and neighbourhood range calculation
def decay(step, max_steps,max_learning_rate,max_m_dsitance):
  coefficient = 1.0 - (np.float64(step)/max_steps)
  learning_rate = coefficient*max_learning_rate
  neighbourhood_range = ceil(coefficient * max_m_dsitance)
  return learning_rate, neighbourhood_range
```

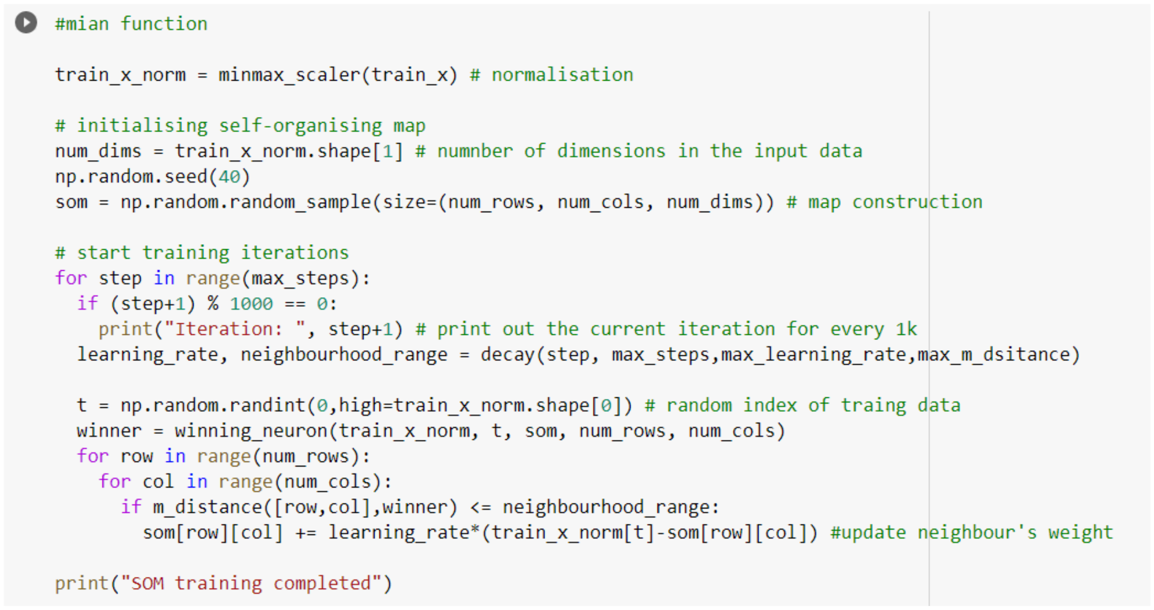
This function returns learning rate and neighborhood range after applying linear decay using the current training step, the maximum number of training steps and maximum neighborhood range and learning rate.

5) Hyper Parameters

```
# hyperparameters
num_rows = 10
num_cols = 10
max_m_dsitance = 4
max_learning_rate = 0.5
max_steps = int(7.5*10e3)
```

Hyper parameters are non-trainable parameters that need to be selected before training algorithms. They are the number of neurons, the dimension of the SOM grid, the number of training steps, the learning rate and the neighborhood range from the BMU. Set the smaller numbers for the grid (10*10).

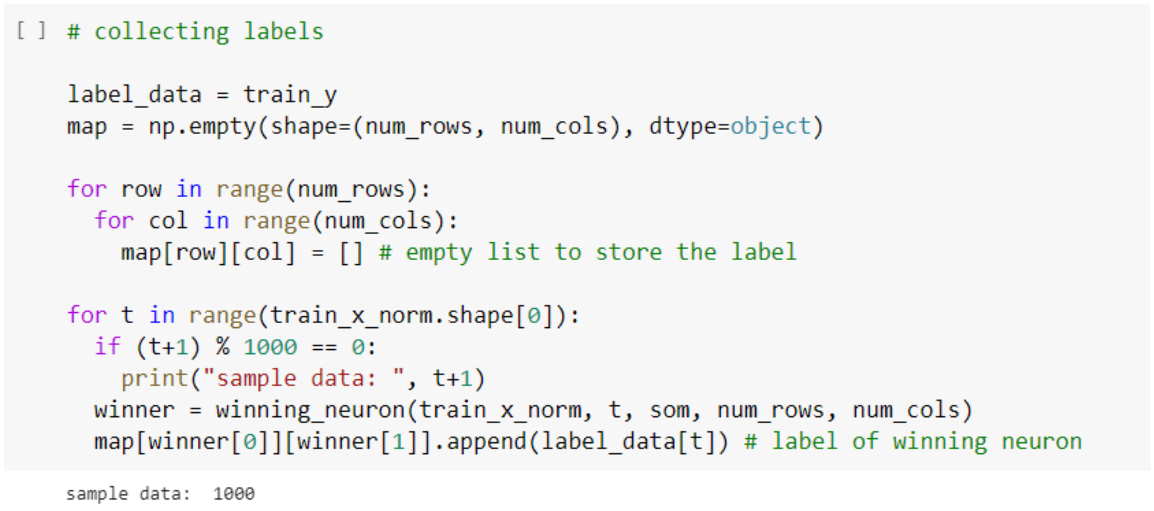
6) Training



```
#mian function

train_x_norm = minmax_scaler(train_x) # normalisation

# initialising self-organising map
num_dims = train_x_norm.shape[1] # numnber of dimensions in the input data
np.random.seed(40)
som = np.random.random_sample(size=(num_rows, num_cols, num_dims)) # map construction

# start training iterations
for step in range(max_steps):
  if (step+1) % 1000 == 0:
    print("Iteration: ", step+1) # print out the current iteration for every 1k
  learning_rate, neighbourhood_range = decay(step, max_steps,max_learning_rate,max_m_dsitance)

  t = np.random.randint(0,high=train_x_norm.shape[0]) # random index of traing data
  winner = winning_neuron(train_x_norm, t, som, num_rows, num_cols)
  for row in range(num_rows):
    for col in range(num_cols):
      if m_distance([row,col],winner) <= neighbourhood_range:
        som[row][col] += learning_rate*(train_x_norm[t]-som[row][col]) #update neighbour's weight

print("SOM training completed")
```

After applying the input data normalization, initialized the map with random values between 0 and 1 for each neuron on the lattice. Then the learning rate and the neighboring range are calculated using the decay function. A sample input observation is randomly selected from the training data and the best matching unit is searched. Based on the Manhattan distance criterion, the neighbors including the winner are selected for learning and weights are adjusted.

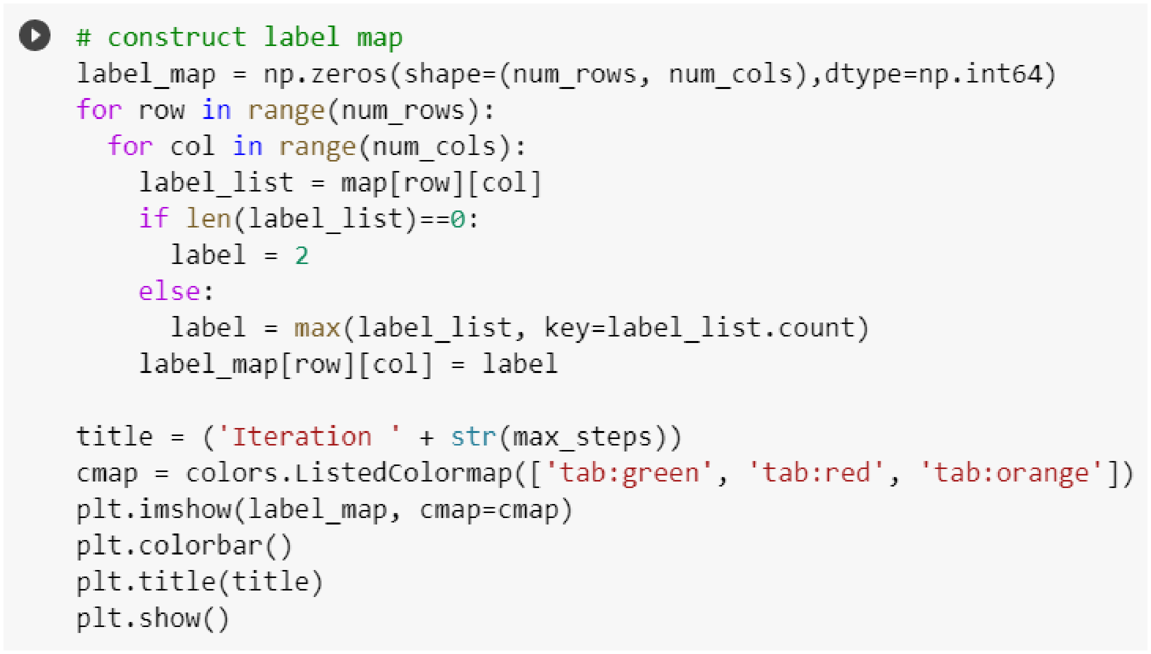
7) Collecting Labels



```
[ ] # collecting labels

    label_data = train_y
    map = np.empty(shape=(num_rows, num_cols), dtype=object)

    for row in range(num_rows):
      for col in range(num_cols):
        map[row][col] = [] # empty list to store the label

    for t in range(train_x_norm.shape[0]):
      if (t+1) % 1000 == 0:
        print("sample data: ", t+1)
      winner = winning_neuron(train_x_norm, t, som, num_rows, num_cols)
      map[winner[0]][winner[1]].append(label_data[t]) # label of winning neuron

    sample data:  1000
```

The labels for each neuron are collected. For each training data, the winning neuron been searched and added the label of the observation to the list for each BMU.

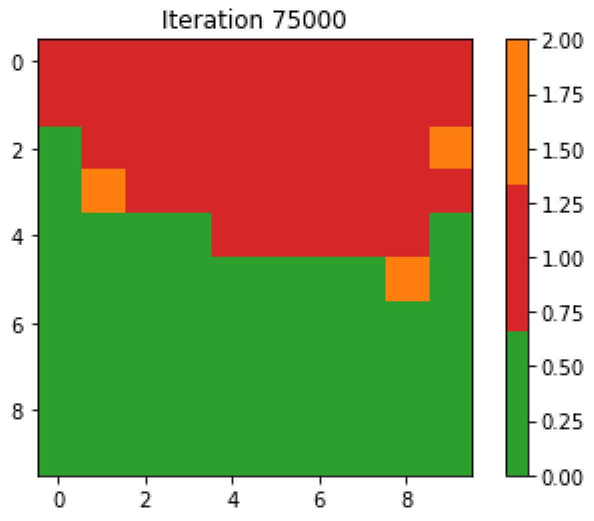
8) Construct Label Map



```
# construct label map
label_map = np.zeros(shape=(num_rows, num_cols),dtype=np.int64)
for row in range(num_rows):
  for col in range(num_cols):
    label_list = map[row][col]
    if len(label_list)==0:
      label = 2
    else:
      label = max(label_list, key=label_list.count)
    label_map[row][col] = label

title = ('Iteration ' + str(max_steps))
cmap = colors.ListedColormap(['tab:green', 'tab:red', 'tab:orange'])
plt.imshow(label_map, cmap=cmap)
plt.colorbar()
plt.title(title)
plt.show()
```

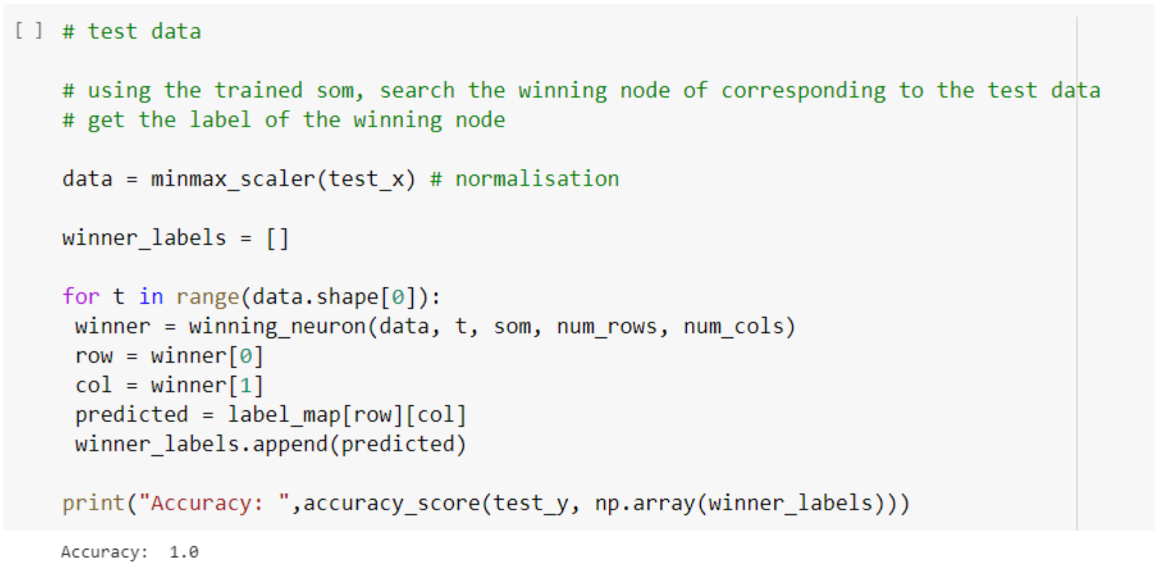
To construct a label map, single label is assigned to each neuron on the map by majority voting. In the case of a neuron where no BMU is selected, the class value 2 is assigned as unidentifiable.

9) Label Map Output



The output shows the region separation between the class 0 and 1 except a couple of cells that do not belong to either class.

10) Test Data



```
[ ] # test data

    # using the trained som, search the winning node of corresponding to the test data
    # get the label of the winning node

    data = minmax_scaler(test_x) # normalisation

    winner_labels = []

    for t in range(data.shape[0]):
     winner = winning_neuron(data, t, som, num_rows, num_cols)
     row = winner[0]
     col = winner[1]
     predicted = label_map[row][col]
     winner_labels.append(predicted)

    print("Accuracy: ",accuracy_score(test_y, np.array(winner_labels)))

    Accuracy:  1.0
```

Finally, conducted a binary classification of the test data using the trained map. Normalized the test x data and search the MBU for each observation t. The label associated with the neuron is returned. The accuracy result was returned with good number.[1]

[1] https://towardsdatascience.com/understanding-self-organising-map-neural-network-with-python-code-7a77f501e985