# PERIYAR UNIVERSITY

**(NAAC 'A++' Grade with CGPA 3.61 (Cycle - 3)**
**State University - NIRF Rank 56 - State Public University Rank 25**
**SALEM - 636 011**

## CENTRE FOR DISTANCE AND ONLINE EDUCATION

## (CDOE)

## MASTER OF COMPUTER APPLICATION

## SEMESTER - II

# SOFT COMPUTING LAB

## (Candidates admitted from 2024 onwards)

# PERIYAR UNIVERSITY

## CENTRE FOR DISTANCE AND ONLINE EDUCATION (CDOE)

## M.C.A 2024 admission onwards

## Soft Computing Lab

Prepared by:

**Centre for Distance and Online Education (CDOE)**
Periyar University
Salem - 636011

## TABLE OF CONTENTS

## 1. IMPLEMENTATION OF LOGICAL GATES USING ARTIFICIAL NEURAL NETWORK

**PROCEDURE:**

Step 1: Start the process

Step 2: Initialize the weights and biases randomly.

Step 3: Feed the input, forward through the network, and calculate the cost.

Step 4: Calculate the gradients of the cost function with respect to the weights and biases.

Step 5: Update the weights and biases using the gradients and a learning rate.

Step 6: Repeat steps 2-4 for a certain number of epochs or until the cost is sufficiently small.

Step 7: Test the neural network by feeding new input through the trained network and make predictions.

Step 8: Evaluate the performance of the network based on the prediction accuracy and adjust the parameters or architecture as necessary.

Step 9: Stop the process.

## 1. Implementation Of Logical Gates Using Artificial Neural Network

**SOURCE CODE**

```
import numpy as np
def perceptron(weights,inputs,bias):
    model=np.add(np.dot(inputs,weights),bias)
    logit=activation_function(model,type="sigmoid")
    return np.round(logit)
def activation_function(model,type="sigmoid"):
    return{
        "sigmoid":1/(1+np.exp(-model))
        }[type]
def compute(data,logic_gate,weights,bias):
    weights=np.array(weights)
    output=np.array([perceptron(weights,datum,bias) for datum in data])
    return output
def print_template(dataset,name,data):
    print("Logic Funtion: {}".format(name.upper()))
    print("X0\tX1\tX2\tY")
    toPrint=["{1}\t{2}\t{3}\t{0}".format(output,*datas) for datas,output in zip(dataset,data)]
    for i in toPrint:
        print(i)
def main():
    dataset=np.array([[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]])
    gates={"and":compute(dataset,"and",[1,1,1],-2),
           "or":compute(dataset,"or",[1,1,1],-0.9),
           "nand":compute(dataset,"nand",[-1,-1,-1],3),
           "nor":compute(dataset,"nor",[-1,-1,-1],1),
           }
    for gate in gates:
        print_template(dataset,gate,gates[gate])
```

```
if __name__=='__main__':
    main()
```

**OUTPUT:**

```
Logic Funtion: AND
X0      X1      X2      Y
0       0       0       0.0
0       0       1       0.0
0       1       0       0.0
0       1       1       0.0
1       0       0       0.0
1       0       1       0.0
1       1       0       0.0
1       1       1       1.0
Logic Funtion: OR
X0      X1      X2      Y
0       0       0       0.0
0       0       1       1.0
0       1       0       1.0
0       1       1       1.0
1       0       0       1.0
1       0       1       1.0
1       1       0       1.0
1       1       1       1.0
```

```
Logic Funtion: NAND
X0      X1      X2      Y
0       0       0       1.0
0       0       1       1.0
0       1       0       1.0
0       1       1       1.0
1       0       0       1.0
1       0       1       1.0
1       1       0       1.0
1       1       1       0.0
Logic Funtion: NOR
X0      X1      X2      Y
0       0       0       1.0
0       0       1       0.0
0       1       0       0.0
0       1       1       0.0
1       0       0       0.0
1       0       1       0.0
1       1       0       0.0
1       1       1       0.0
```

**RESULT**

Thus the above program has been executed successfully.

## 2. IMPLEMENTATION OF PERCEPTRON ALGORITHM

**PROCEDURE:**

Step 1: Start the process

Step 2: Define the perceptron function with weights, inputs, and bias as inputs.

Step 3: Compute the model as the dot product of inputs and weights, plus the bias.

Step 4: Compute the output of the activation function (sigmoid) applied to the model.

Step 5: Round the output to get the final predicted value.

Step 6: Define the activation function with model and type (default: sigmoid) as inputs.

Step 7: Define the compute function with data, logic_gate, weights, and bias as inputs.

Step 8: Define the dataset, compute the logic gates (AND, OR, NAND, NOR) using the compute function, and print the results.

Step 9: Stop the process.

**SOURCE CODE**

```
import numpy as np
from matplotlib import pyplot as plt
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
def initializeParameters(inputFeatures, neuronsInHiddenLayers, outputFeatures):
    W1 = np.random.randn(neuronsInHiddenLayers, inputFeatures)
    W2 = np.random.randn(outputFeatures, neuronsInHiddenLayers)
    b1 = np.zeros((neuronsInHiddenLayers, 1))
    b2 = np.zeros((outputFeatures, 1))

    parameters = {"W1" : W1, "b1": b1,
                  "W2" : W2, "b2": b2}
    return parameters
def forwardPropagation(X, Y, parameters):
    m = X.shape[1]
    W1 = parameters["W1"]
    W2 = parameters["W2"]
    b1 = parameters["b1"]
    b2 = parameters["b2"]
    Z1 = np.dot(W1, X) + b1
    A1 = sigmoid(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)
    cache = (Z1, A1, W1, b1, Z2, A2, W2, b2)
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), (1 - Y))
    cost = -np.sum(logprobs) / m
    return cost, cache, A2
```

```
def backwardPropagation(X, Y, cache):
    m = X.shape[1]
    (Z1, A1, W1, b1, Z2, A2, W2, b2) = cache
    dZ2 = A2 - Y
    dW2 = np.dot(dZ2, A1.T) / m
    db2 = np.sum(dZ2, axis = 1, keepdims = True)
    dA1 = np.dot(W2.T, dZ2)
    dZ1 = np.multiply(dA1, A1 * (1- A1))
    dW1 = np.dot(dZ1, X.T) / m
    db1 = np.sum(dZ1, axis = 1, keepdims = True) / m
    gradients = {"dZ2": dZ2, "dW2": dW2, "db2": db2,
                 "dZ1": dZ1, "dW1": dW1, "db1": db1}
    return gradients
def updateParameters(parameters, gradients, learningRate):
    parameters["W1"] = parameters["W1"] - learningRate * gradients["dW1"]
    parameters["W2"] = parameters["W2"] - learningRate * gradients["dW2"]
    parameters["b1"] = parameters["b1"] - learningRate * gradients["db1"]
    parameters["b2"] = parameters["b2"] - learningRate * gradients["db2"]
    return parameters
X = np.array([[0, 0, 1, 1], [0, 1, 0, 1]]) # XOR input
Y = np.array([[0, 1, 1, 0]]) # XOR output
neuronsInHiddenLayers = 2 # number of hidden layer neurons (2)
inputFeatures = X.shape[0] # number of input features (2)
outputFeatures = Y.shape[0] # number of output features (1)
parameters = initializeParameters(inputFeatures, neuronsInHiddenLayers,
outputFeatures)
epoch = 100000
learningRate = 0.01
losses = np.zeros((epoch, 1))
```

```
for i in range(epoch):
        losses[i, 0], cache, A2 = forwardPropagation(X, Y, parameters)
        gradients = backwardPropagation(X, Y, cache)
        parameters = updateParameters(parameters, gradients, learningRate)
plt.figure()
plt.plot(losses)
plt.xlabel("EPOCHS")
plt.ylabel("Loss value")
plt.show()
X = np.array([[1, 1, 0, 0], [0, 1, 0, 1]]) # XOR input
cost, _, A2 = forwardPropagation(X, Y, parameters)
prediction = (A2 > 0.5) * 1.0
print(prediction)
```

## OUTPUT

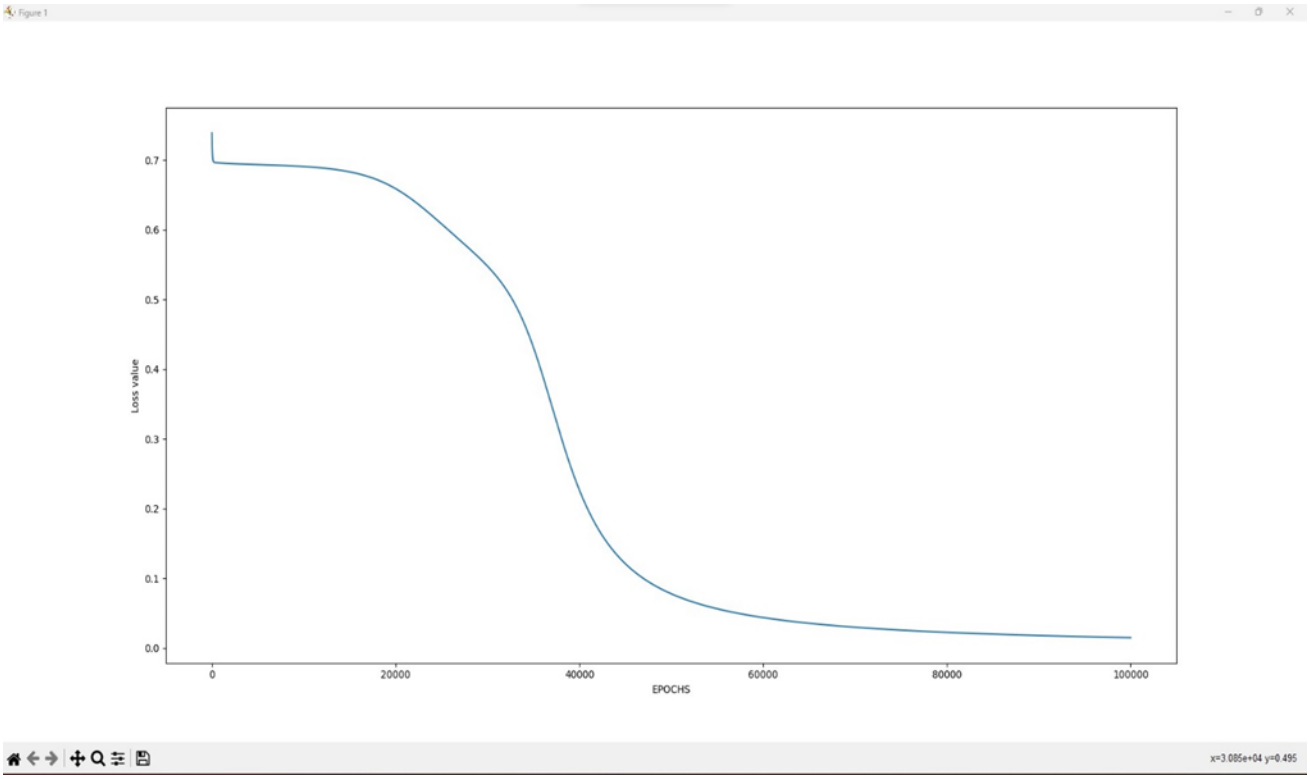


Figure 1
Loss value
EPOCHS
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
20000
40000
60000
80000
100000
x=3.085e+04 y=0.495

**RESULT**

Thus the above program has been executed successfully.

## 3. IMPLEMENTATION OF BACK PROPAGATION ALGORITHM

**PROCEDURE:**

Step 1: Start the process

Step 2: Define input and output data

Step 3: Normalize the input data

Step 4: Compute the value of sigmoid activation function and its derivative

Step 5: Set the number of epochs and learning rate

Step 6: Define the number of neurons in the input, hidden, and output layers

Step 7: Initialize weights and biases for the hidden and output layers

Step 8: Implement forward propagation and calculate the predicted output

Step 9: Implement backpropagation and update the weights and biases

Step 10: Repeat steps 7-8 for the specified number of epochs

Step 11: Print the input data, actual output, and predicted output for each epoch

Step 12: Print the final input data, actual output, and predicted output

Step 13: Stop the process

**SOURCE CODE:**

```
import numpy as np
X = np.array(([2, 9], [1, 5], [3, 6]), dtype=float)
y = np.array(([92], [86], [89]), dtype=float)
X = X/np.amax(X,axis=0) #maximum of X array longitudinally
y = y/100
def sigmoid (x):
    return 1/(1 + np.exp(-x))
def derivatives_sigmoid(x):
    return x * (1 - x)
epoch=5
lr=0.1 #Setting learning rate
inputlayer_neurons = 2
hiddenlayer_neurons = 3 output_neurons = 1
wh=np.random.uniform(size=(inputlayer_neurons,hiddenlayer_neurons))
bh=np.random.uniform(size=(1,hiddenlayer_neurons))
wout=np.random.uniform(size=(hiddenlayer_neurons,output_neurons))
bout=np.random.uniform(size=(1,output_neurons))
for i in range(epoch):
    hinp1=np.dot(X,wh)
    hinp=hinp1 + bh
    hlayer_act = sigmoid(hinp)
    outinp1=np.dot(hlayer_act,wout)
    outinp= outinp1+bout
    output = sigmoid(outinp)
   EO = y-output
    outgrad = derivatives_sigmoid(output)
    d_output = EO * outgrad
    EH = d_output.dot(wout.T)
```

```
    hiddengrad = derivatives_sigmoid(hlayer_act)
    d_hiddenlayer = EH * hiddengrad
    wout += hlayer_act.T.dot(d_output) *lr
    wh += X.T.dot(d_hiddenlayer) *lr
    print ("-----------Epoch-", i+1, "Starts----------")
    print("Input: \n" + str(X))
    print("Actual Output: \n" + str(y))
    print("Predicted Output: \n" ,output)
    print ("-----------Epoch-", i+1, "Ends----------\n")
print("Input: \n" + str(X))
print("Actual Output: \n" + str(y))
print("Predicted Output: \n" ,output)
```

**OUTPUT**

```
----------Epoch- 1 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.87482092]
 [0.86227273]
 [0.87952658]]
----------Epoch- 1 Ends----------

----------Epoch- 2 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.87494108]
 [0.86238964]
 [0.87964519]]
----------Epoch- 2 Ends----------
```

```
----------Epoch- 3 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.87506025]
 [0.86250561]
 [0.87976283]]
----------Epoch- 3 Ends----------

----------Epoch- 4 Starts----------
Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.87517846]
 [0.86262065]
 [0.8798795 ]]
----------Epoch- 4 Ends----------
```

```
----------Epoch- 5 Ends----------

Input:
[[0.66666667 1.        ]
 [0.33333333 0.55555556]
 [1.         0.66666667]]
Actual Output:
[[0.92]
 [0.86]
 [0.89]]
Predicted Output:
 [[0.8752957 ]
 [0.86273476]
 [0.87999523]]
```

**RESULT**

Thus the above program has been executed successfully.

## 4. IMPLEMENTATION OF SELF ORGANIZING MAPS

**PROCEDURE:**

Step 1: Start the process

Step 2: Import necessary libraries such as numpy, matplotlib, pandas, sklearn, and minisom

Step 3: Load the dataset using pandas.

Step 4: Prepare the data: Preprocess the data by scaling it using MinMaxScaler and splitting it into input and output variables.

Step 5: Train the self-organizing map by randomly initializing the weights and training it on the data for a certain number of iterations.

Step 6: Find the nodes with the highest mean distance to their neighboring nodes on the self-organizing map and inverse transforming the data.

Step 7: Stop the process.

**SOURCE CODE:**

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
dataset = pd.read_csv(Z:\soft\Credit_Card_Applications.csv')
dataset.head(10)
dataset.dtypes
x = dataset.iloc[:,:-1].values
y = dataset.iloc[:,-1].values
from sklearn.preprocessing import MinMaxScaler
sc = MinMaxScaler(feature_range=(0, 1))
X = sc.fit_transform(X)
from minisom import MiniSom
som = MiniSom(x=10,y=10,input_len=15)
som.random_weights_init(X)
som.train_random(data = X, num_iteration = 100)
som.distance_map().shape
from pylab import bone, colorbar, pcolor, plot, show
bone()
pcolor(som.distance_map().T)
colorbar()
bone()
pcolor(som.distance_map().T)
colorbar()
markers = ['o', 's']
colors = ['r', 'g']
for i, x in enumerate(X):
    w = som.winner(x)
```

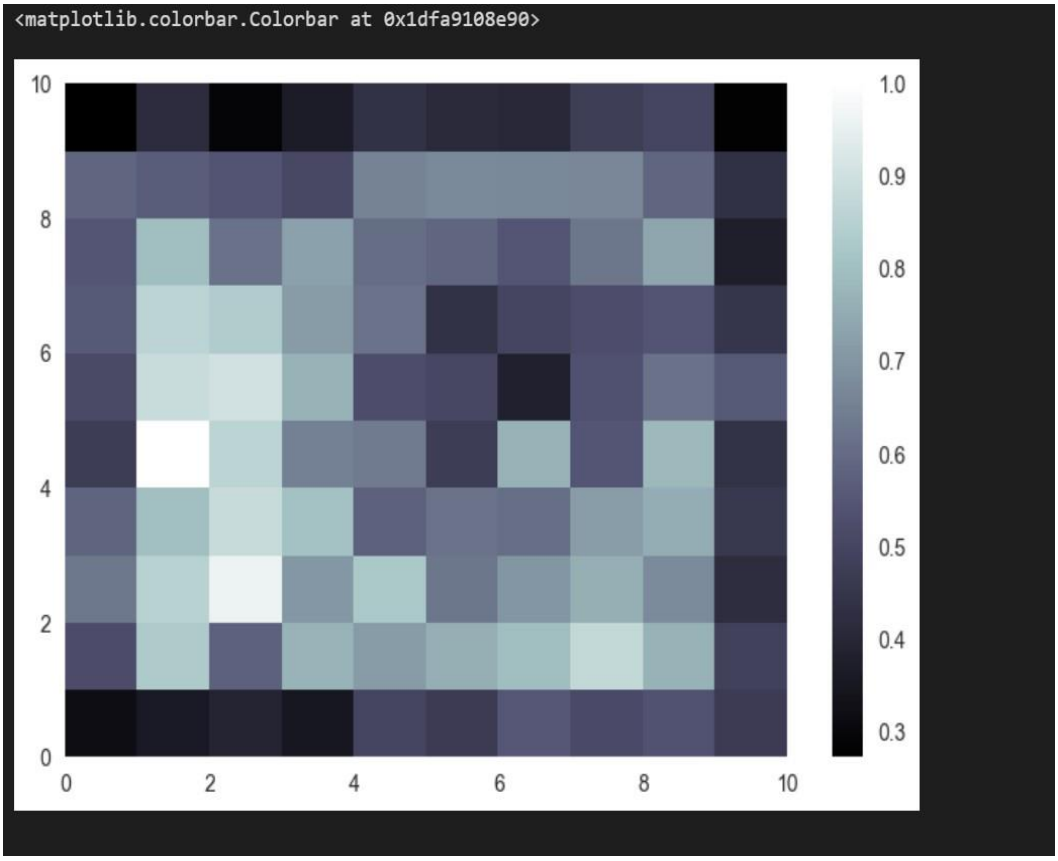
```
    plot(w[0] + 0.5,
         w[1] + 0.5,
         markers[y[i]],
         markeredgecolor = colors[y[i]],
         markerfacecolor = 'None',
         markersize = 10,
         markeredgewidth = 2)
show()
mappings = som.win_map(X)
mappings.keys()
frauds = np.concatenate((mappings[(6,4)],mappings[(6,8)]),axis=0)
np.asarray(frauds).shape
frauds = sc.inverse_transform(frauds)
np.asarray(frauds).shape
print('Fraud Customer IDs')
for i in frauds[:, 0]:
    print(int(i))
```

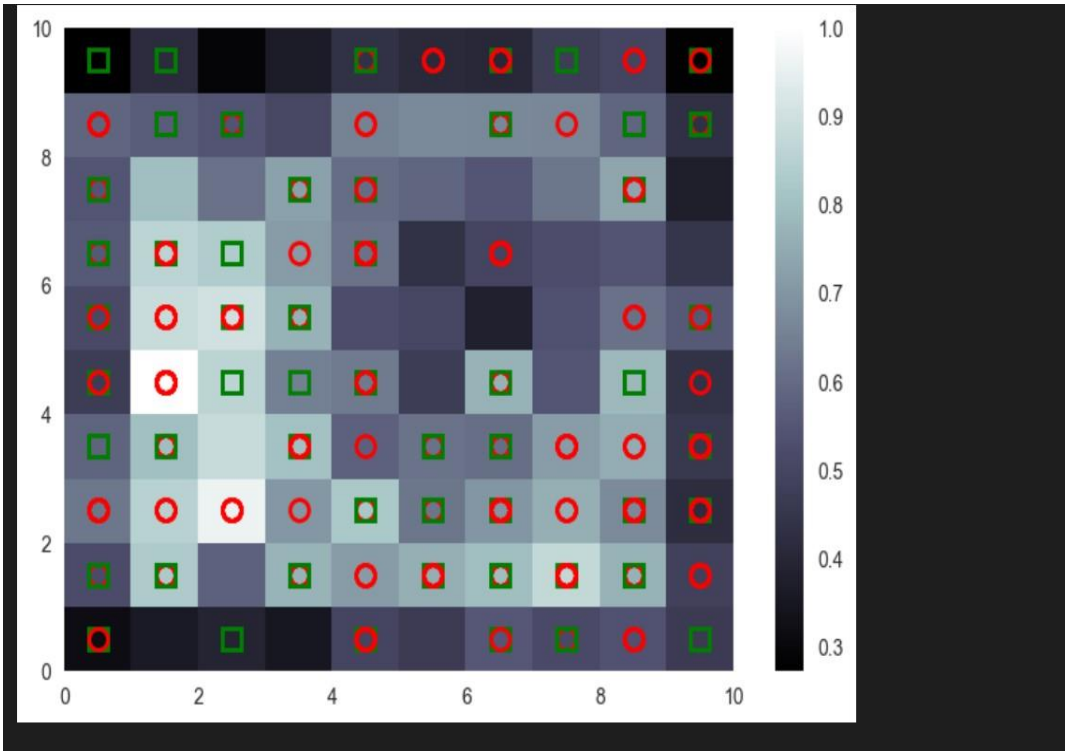
**OUTPUT:**

| | CustomerID | A1 | A2 | A3 | A4 | A5 | A6 | A7 | A8 | A9 | A10 | A11 | A12 | A13 | A14 | Class |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 15776156 | 1 | 22.08 | 11.460 | 2 | 4 | 4 | 1.585 | 0 | 0 | 0 | 1 | 2 | 100 | 1213 | 0 |
| 1 | 15739548 | 0 | 22.67 | 7.000 | 2 | 8 | 4 | 0.165 | 0 | 0 | 0 | 0 | 2 | 160 | 1 | 0 |
| 2 | 15662854 | 0 | 29.58 | 1.750 | 1 | 4 | 4 | 1.250 | 0 | 0 | 0 | 1 | 2 | 280 | 1 | 0 |
| 3 | 15687688 | 0 | 21.67 | 11.500 | 1 | 5 | 3 | 0.000 | 1 | 1 | 11 | 1 | 2 | 0 | 1 | 1 |
| 4 | 15715750 | 1 | 20.17 | 8.170 | 2 | 6 | 4 | 1.960 | 1 | 1 | 14 | 0 | 2 | 60 | 159 | 1 |
| 5 | 15571121 | 0 | 15.83 | 0.585 | 2 | 8 | 8 | 1.500 | 1 | 1 | 2 | 0 | 2 | 100 | 1 | 1 |
| 6 | 15726466 | 1 | 17.42 | 6.500 | 2 | 3 | 4 | 0.125 | 0 | 0 | 0 | 0 | 2 | 60 | 101 | 0 |
| 7 | 15660390 | 0 | 58.67 | 4.460 | 2 | 11 | 8 | 3.040 | 1 | 1 | 6 | 0 | 2 | 43 | 561 | 1 |
| 8 | 15663942 | 1 | 27.83 | 1.000 | 1 | 2 | 8 | 3.000 | 0 | 0 | 0 | 0 | 2 | 176 | 538 | 0 |
| 9 | 15638610 | 0 | 55.75 | 7.080 | 2 | 4 | 8 | 6.750 | 1 | 1 | 3 | 1 | 2 | 100 | 51 | 0 |

```
CustomerID      int64
A1              int64
A2            float64
A3            float64
A4              int64
A5              int64
A6              int64
A7            float64
A8              int64
A9              int64
A10             int64
A11             int64
A12             int64
A13             int64
A14             int64
Class           int64
dtype: object
```

```
(10, 10)
```


<matplotlib.colorbar.Colorbar at 0x1dfa9108e90>
0
2
4
6
8
10
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0


0
2
4
6
8
10
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0

```
dict_keys([(3, 3), (7, 1), (9, 3), (8, 7), (1, 9), (7, 9), (4, 0), (6, 6), (2, 0), (0, 9), (3, 1), (6, 0), (5, 1), (2, 8), (9, 8), (6, 1), (6, 3),
           (8, 1), (7, 8), (4, 4), (1, 5), (0, 2), (1, 6), (1, 8), (5, 3), (8, 2), (6, 8), (2, 5), (6, 2), (6, 4), (8, 4), (5, 9), (1, 2), (1, 4), (1, 1),
           (2, 4), (3, 5), (9, 4), (4, 9), (0, 7), (7, 3), (0, 6), (0, 5), (2, 2), (9, 2), (4, 6), (7, 0), (4, 2), (3, 7), (9, 0), (6, 9), (9, 9), (2, 6),
           (8, 3), (0, 0), (3, 6), (0, 4), (7, 2), (0, 1), (4, 1), (9, 1), (8, 9), (9, 5), (3, 2), (3, 4), (8, 5), (8, 0), (4, 7), (0, 3), (8, 8), (5, 2),
           (1, 3), (0, 8), (4, 8), (4, 3)])
```

```
(23, 15)
```

```
(23, 15)
```

```
Fraud Customer IDs
15731586
15575605
15644453
15699238
15704581
15725776
15815040
15697460
15737998
15586183
15718921
15607988
15683993
15797767
15744044
15632010
15804235
15627365
15700300
15791944
15576680
15620570
15592412
```

**RESULT**

Thus the above program has been executed successfully.

## 5. IMPLEMENTATION OF RADIAL BASIS FUNCTION NETWORK

**PROCEDURE:**

Step 1: Start the process.

Step 2: Initialize the RBFNN class with random weights, biases, and define parameters like kernels, centers, beta, learning rate, and epochs.

Step 3: Define the forward propagation process using the radial basis function (RBF) and a linear activation function to compute the output.

Step 4: Calculate the error using the least-square error function between the predicted output and the true value.

Step 5: Implement backpropagation to adjust the weights and biases using gradient descent.

Step 6: Train the model using the fit function over a set number of epochs with the XOR gate data as input, updating weights and biases in each epoch.

Step 7: Test the trained model using the predict function to evaluate XOR logic and print the results.

**SOURCE CODE:**

```
import numpy as np
class RBFNN:
    def __init__(self, kernels,centers, beta=1,lr=0.1,epochs=80) -> None
        self.kernels = kernels
        self.centers = centers
        self.beta = beta
        self.lr = lr
        self.epochs = epochs
        self.W = np.random.randn(kernels,1)
        self.b = np.random.randn(1,1)
        self.errors = []
        self.gradients = []
    def rbf_activation(self,x,center):
        return np.exp(-self.beta*np.linalg.norm(x - center)**2)
    def linear_activation(self,A):
        return self.W.T.dot(A) + self.b
    def least_square_error(self,pred,y):
        return (y - pred).flatten()**2
    def _forward_propagation(self,x):
        a1 = np.array([
            [self.rbf_activation(x,center)]
            for center in self.centers
        ])
        a2 = self.linear_activation(a1)
        return a2, a1
    def _backpropagation(self, y, pred,a1):
                dW = -(y - pred).flatten()*a1
        db = -(y - pred).flatten()
```

```
        self.W = self.W -self.lr*dW
        self.b = self.b -self.lr*db
        return dW, db
    def fit(self,X,Y):
        for _ in range(self.epochs):
            for x,y in list(zip(X,Y)):
                pred, a1 = self._forward_propagation(x)
                error = self.least_square_error(pred[0],y[0,np.newaxis])
                self.errors.append(error)
                dW, db = self._backpropagation(y,pred,a1)
                self.gradients.append((dW,db))
    def predict(self,x):
        a2,a1 = self._forward_propagation(x)
        return 1 if np.squeeze(a2) >= 0.5 else 0
def main():
        X = np.array([
            [0,0],
            [0,1],
            [1,0],
            [1,1]
        ])
        Y = np.array([
            [0],
            [1],
            [1],
            [0]
        ])
        rbf = RBFNN(kernels=2,
```

```
            centers=np.array([
                [0,1],
                [1,0]

            ]),
            beta=1,
            lr= 0.1,
            epochs=80
            )
        rbf.fit(X,Y)
        print(f"RBFN weights : {rbf.W}")
        print(f"RBFN bias : {rbf.b}")
        print()
        print("-- XOR Gate --")
        print(f"| 1 xor 1 : {rbf.predict(X[3])} |")
        print(f"| 0 xor 0 : {rbf.predict(X[0])} |")
        print(f"| 1 xor 0 : {rbf.predict(X[2])} |")
        print(f"| 0 xor 1 : {rbf.predict(X[1])} |")
        print("_________________")
if __name__ == "__main__":
    main()
```

**OUTPUT:**

```
RBFN weights : [[1.0423964 ]
 [0.99528412]]
RBFN bias : [[-0.42998542]]

-- XOR Gate --
| 1 xor 1 : 0 |
| 0 xor 0 : 0 |
| 1 xor 0 : 1 |
| 0 xor 1 : 1 |

_________________
```

**RESULT**

Thus the above program has been executed successfully.

## 6. IMPLEMENTATION OF DE-MORGAN's LAW

**PROCEDURE:**

Step 1: Start the process

Step 2: Import the necessary modules: numpy.

Step 3: Set the random seed to ensure reproducibility of results.

Step 4: Generate a random input vector and weight vector.

Step 5: Calculate the dot product of the input and weight vectors.

Step 6: Define a function for the linear threshold gate with a specified threshold value.

Step 7: Test the function with different threshold values.

Step 8: Create an input table with various combinations of binary inputs.

Step 9: Define a weight vector.

Step 10: Calculate the dot product of the input table and weight vector.

Step 11: Use the linear threshold function to generate binary outputs for each row in the input table based on a specified threshold value.

**SOURCE CODE:**

```
#De Morgan's Law 1: ~(A U B) = ~A ∩ ~B
#De Morgan's Law 2: ~(A ∩ B) = ~A U ~B
S1 = set([x for x in range(31) if x%3==0])
print ("Set S1:", S1)
S2 = set([x for x in range(31) if x%5==0])
print ("Set S2:", S2)
S_difference = S2-S1
print("Difference of S1 and S2 i.e. S2\S1:", S_difference)
S_difference = S1.difference(S2)
print("Difference of S2 and S1 i.e. S1\S2:", S_difference)
print("S1",S1)
print("S2",S2)
print("Symmetric difference", S1^S2)
print("Symmetric difference", S2.symmetric_difference(S1))
def de_morgan_1(A, B):
    return [not x for x in [not a or not b for a, b in zip(A, B)]]
def de_morgan_2(A, B):
    return [not x for x in [not a and not b for a, b in zip(A, B)]]
A = [True, False, True, False]
B = [True, True, False, False]
print(de_morgan_1(A, B))
print(de_morgan_2(A, B))
```

**OUTPUT:**

```
Set S1: {0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30}
Set S2: {0, 5, 10, 15, 20, 25, 30}
Difference of S1 and S2 i.e. S2\S1: {25, 10, 20, 5}
Difference of S2 and S1 i.e. S1\S2: {3, 6, 9, 12, 18, 21, 24, 27}
S1 {0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30}
S2 {0, 5, 10, 15, 20, 25, 30}
Symmetric difference {3, 5, 6, 9, 10, 12, 18, 20, 21, 24, 25, 27}
Symmetric difference {3, 5, 6, 9, 10, 12, 18, 20, 21, 24, 25, 27}
[True, False, False, False]
[True, True, True, False]
[ True False False False]
[ True  True  True False]
[[False, False, False], [False, False, False]]
[[True, True, True], [True, True, True]]
[[0.4, 0.5, 0.6], [0.6, 0.5, 0.4]]
[[0.9, 0.8, 0.7], [0.7, 0.8, 0.9]]
```

**RESULT**

Thus the above program has been executed successfully.

## 7. IMPLEMENTATION OF Mc CULLOCH PITS ARTIFICIAL NEURON MODEL

**PROCEDURE:**

Step 1: Start the process

Step 2: Initialize the weights and biases randomly

Step 3: Feed the input through the network and compute the output

Step 4: Calculate the loss between the predicted output and the true output

Step 5: Update the weights and biases using gradient descent to minimize the loss

Step 6: Repeat steps 2-4 for a number of epochs or until the loss is minimized to a satisfactory level

Step 7: Stop the process.

**SOURCE CODE:**

```
import numpy as np
np.random.seed(seed=0)
I = np.random.choice([0,1], 3)# generate random vector I, sampling from {0,1}
W = np.random.choice([-1,1], 3) # generate random vector W, sampling from {-1,1}
print(f'Input vector:{I}, Weight vector:{W}')
dot = I @ W
print(f'Dot product: {dot}')
def linear_threshold_gate(dot: int, T: float) -> int:
    '''Returns the binary threshold output'''
    if dot >= T:
        return 1
    else:
        return 0
T = 1
activation = linear_threshold_gate(dot, T)
print(f'Activation: {activation}')
T = 3
activation = linear_threshold_gate(dot, T)
print(f'Activation: {activation}')
input_table = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
])

print(f'input table:\n{input_table}')
```

```
weights = np.array([1,1])
print(f'weights: {weights}')
dot_products = input_table @ weights
print(f'Dot products: {dot_products}')
T = 2
for i in range(0,4):
    activation = linear_threshold_gate(dot_products[i], T)
    print(f'Activation: {activation}')
input_table = np.array([
    [0,0],
    [1,1],
    [0,1],
    [1,1]
])
print(f'input table:\n{input_table}')
weights = np.array([1,1])
print(f'weights: {weights}')
dot_products = input_table @ weights
print(f'Dot products: {dot_products}')
T = 1
for i in range(0,4):
    activation = linear_threshold_gate(dot_products[i], T)
    print(f'Activation: {activation}')
input_table = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
```

```
])
print(f'input table:\n{input_table}')
input_table = np.array([
    [0,0],
    [0,1],
    [1,0],
    [1,1]
])
print(f'input table:\n{input_table}')
dot_products = input_table @ weights
print(f'Dot products: {dot_products}')
T = 0
for i in range(0,4):
    activation = linear_threshold_gate(dot_products[i], T)
    print(f'Activation: {activation}')
num_ip = int(input("Enter the number of inputs : "))
w1 = 1
w2 = 1
print("For the ", num_ip , " inputs calculate the net input using yin = x1w1 + x2w2 ")
x1 = []
x2 = []
for j in range(0, num_ip):
        ele1 = int(input("x1 = "))
        ele2 = int(input("x2 = "))
        x1.append(ele1)
        x2.append(ele2)
        print("x1 = ",x1)
        print("x2 = ",x2)
```

```
        n = x1 * w1
        m = x2 * w2
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] + m[i])
    print("Yin = ",Yin)
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] - m[i])
    print("After assuming one weight as excitatory and the other as inhibitory Yin = ",Yin)
Y=[]
for i in range(0, num_ip):
        if(Yin[i]>=1):
            ele= 1
            Y.append(ele)
        if(Yin[i]<1):
            ele= 0
            Y.append(ele)
            print("Y = ",Y)
```

**OUTPUT:**

```
Input vector:[0 1 1], Weight vector:[-1  1  1]
Dot product: 2
Activation: 1
Activation: 0
input table:
[[0 0]
 [0 1]
 [1 0]
 [1 1]]
weights: [1 1]
Dot products: [0 1 1 2]
Activation: 0
Activation: 0
Activation: 0
Activation: 1
input table:
[[0 0]
 [1 1]
 [0 1]
 [1 1]]
weights: [1 1]
x2 = 1
x1 =  [2]
x2 =  [1]
x1 = 2
x2 = 2
x1 =  [2, 2]
x2 =  [1, 2]
Yin =  [3]
Yin =  [3, 4]
After assuming one weight as excitatory and the other as inhibitory Yin =  [1]
After assuming one weight as excitatory and the other as inhibitory Yin =  [1, 0]
Y =  [1, 0]
```

**RESULT**

Thus the above program has been executed successfully.

## 8. IMPLEMENTATION OF SIMPLE GENETIC ALGORITHM

**PROCEDURE:**

```
import random
def fitness_function(x):
    return x**2
def genetic_algorithm(population_size, num_generations, mutation_rate):
    population = [random.uniform(-10, 10) for _ in range(population_size)]
    for generation in range(num_generations):
        fitness_scores = [fitness_function(x) for x in population]
        parents = []
        for i in range(population_size // 2):
            ind1 = random.choice(population)
            ind2 = random.choice(population)
            parent = ind1 if fitness_function(ind1) > fitness_function(ind2) else ind2
            parents.append(parent)
                new_population = []
        for parent in parents:
            if random.random() < mutation_rate:
                offspring = parent + random.uniform(-1, 1)
            else:
                offspring = parent
            new_population.append(offspring)
                population = new_population
    return max(population, key=fitness_function)
best_individual = genetic_algorithm(population_size=78, num_generations=30, mutation_rate=0.3)
print(f"The best individual is {best_individual} with a fitness of {fitness_function(best_individual)}")
```

**OUTPUT:**

```
The best individual is 28.579085099763702 with a fitness of 816.7641051395357
```

**RESULT**

Thus the above program has been executed successfully.

## 9. IMPLEMENTATION OF FUZZY BASED LOGICAL OPERATIONS

**PROCEDURE:**

Step 1: Start the process

Step 2: Initialize the population

Step 3: Evaluate the fitness of each individual

Step 4: Select parents for reproduction

Step 5: Generate offspring through genetic operators

Step 6: Replace the old population with the new one

Step 7: Stop the process

**SOURCE CODE :**

```
import numpy as np
def union(A,B):
    result={}
    for i in A:
        if(A[i]>B[i]):
            result[i]=A[i]
        else:
            result[i]=B[i]
    print("Union of two sets is",result)

def intersection(A,B):
    result={}
    for i in A:
        if(A[i]<B[i]):
            result[i]=A[i]
        else:
            result[i]=B[i]
    print("Intersection of two sets is",result)

def complement(A,B):
    result={}
    result1={}
    for i in A:
        result[i]=round(1-A[i],2)
    for i in B:
        result1[i]=round(1-B[i],2)
    print("Complement of  1st set is",result)
```

```
    print("Complement of  2nd set is",result1)

def difference(A,B):
    result={}
    for i in A:
        result[i]=round(min(A[i],1-B[i]),2)
    print("Difference of two sets is",result)

def main():

    while True:
        print("Menu Driven Program")
        print("1.Union")
        print("2.Intersection")
        print("3.Complement")
        print("4.Difference")

        choice=int(input("Enter your choice:"))
        if choice==1:
            union(d,d1)

        elif choice==2:
            intersection(d,d1)

        elif choice==3:
            complement(d,d1)

        elif choice==4:
```

```
            difference(d,d1)
        elif choice==7:
            break
        else:
            print("Wrong choice")


if __name__ == "__main__":
    print("----------------------"+
        "FUZZY SET OPERATIONS"+
        "----------------------")
    n = int(input("enter  no.of elements of set 1:"))
    d = {}
    for i in range(n):
        keys = input()
        values = float(input())
        d[keys] = values
    n1 = int(input("enter  no.of elements of set 2:"))
    d1 = {}
    for i in range(n1):
        keys1 = input()
        values1 = float(input())
        d1[keys1] = values1
main()
```

**OUTPUT:**

```
----------------------FUZZY SET OPERATIONS----------------------
Menu Driven Program
1.Union
2.Intersection
3.Complement
4.Difference
5.Exit
Union of two sets is {'1': 0.4, '2': 0.7}
Menu Driven Program
1.Union
2.Intersection
3.Complement
4.Difference
5.Exit
Intersection of two sets is {'1': 0.3, '2': 0.4}
Menu Driven Program
1.Union
2.Intersection
3.Complement
4.Difference
5.Exit
Complement of  1st set is {'1': 0.6, '2': 0.3}
Complement of  2nd set is {'1': 0.7, '2': 0.6}
Menu Driven Program
1.Union
...
2.Intersection
3.Complement
4.Difference
5.Exit
```

**RESULT**

Thus the above program has been executed successfully.

## 10. IMPLEMENTATION OF FUZZY BASED ARITHMETIC OPERATIONS

**PROCEDURE:**

Step 1: Start the process

Step 2: Define three functions for performing fuzzy addition, subtraction, and multiplication operations.

Step 3: Each function takes two lists a and b as input and returns a new list that represents the result of the corresponding operation.

Step 4: Initialize two lists a and b with some values.

Step 5: Call the fuzzy_addition, subtraction, multiplication functions with lists a and b as input and store the result in new lists c, d, and e.

Step 6: Print the value of c, d, and e.

Step 7: End the program.

**SOURCE CODE:**

```
def fuzzy_addition(a, b):
    result = []
    for i in range(len(a)):
        result.append(max(a[i],b[i]))
    return result
def fuzzy_substraction(a, b):
    result = []
    for i in range(len(a)):
        result.append(max(0, a[i]-b[i]))
    return result
def fuzzy_multiplication(a, b):
    result = []
    for i in range(len(a)):
        result.append(min(a[i],b[i]))
    return result
a = [0.2, 0.4, 0.6]
b = [0.3, 0.5, 0.6]
c = fuzzy_addition(a, b)
print(c)
d = fuzzy_substraction(a, b)
print(d)
e = fuzzy_multiplication(a, b)
print(e)
```

**OUTPUT:**

```
[0.3, 0.5, 0.6]
[0, 0, 0]
[0.2, 0.4, 0.6]
```

**RESULT**

Thus the above program has been executed successfully.