MACHINE LEARNING LAB EXERCISE

## DATE: 06-02-2025

**QUESTIONS:  
Lab Exercise: Understanding the Perceptron Logically (Note: functions should not be called directly)**

**· Understand the working of a perceptron model through logical implementation.**

**· Implement a perceptron to classify linearly separable data.**

**1. Implementing Perceptron from Scratch: Implement a single-layer perceptron using Python (NumPy). Use a simpl AND gate or OR gate.**

**o Initialize weights and bias randomly.**

**o Use the sigmoid function as the activation function.**

**o Update weights using the perceptron learning rule or with guess**

**o Train the perceptron and test**

ANSWER:

CODE:  
import numpy as np

import matplotlib.pyplot as plt

#this is the activation function that i am using

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# the derivative of the activation function is taken for the back propagation

def gradient\_descent(x):

return x \* (1 - x)

class Perceptron:

def \_\_init\_\_(self, input\_size, learning\_rate=0.1):

self.weights = np.random.rand(input\_size)

self.bias = np.random.rand(1)

self.learning\_rate = learning\_rate

def predict(self, inputs):

linear\_output = np.dot(inputs, self.weights) + self.bias

return sigmoid(linear\_output)

def train(self, inputs, targets, epochs=100000):

for epoch in range(epochs):

for x, target in zip(inputs, targets):

prediction = self.predict(x)

error = target - prediction

self.weights += self.learning\_rate \* error \* x \* gradient\_descent(prediction)

self.bias += self.learning\_rate \* error \* gradient\_descent(prediction)

x\_vector = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y\_and = np.array([0, 0, 0, 1])

y\_or = np.array([0, 1, 1, 1])

input\_size = 2

epochs = 100000

perceptron\_and = Perceptron(input\_size)

perceptron\_and.train(x\_vector, y\_and, epochs)

print("AND GATE PREDICTION")

for x, y in zip(x\_vector,y\_and):

print(f"Input: {x} -> Predicted: {perceptron\_and.predict(x)} -> Actual : {y}")

perceptron\_or = Perceptron(input\_size)

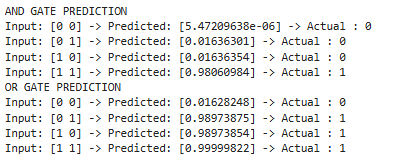
perceptron\_or.train(x\_vector, y\_or, epochs)

print("OR GATE PREDICTION")

for x, y in zip(x\_vector,y\_or):

print(f"Input: {x} -> Predicted: {perceptron\_or.predict(x)} -> Actual : {y}")

OUTPUT



**2. Visualizing the Decision Boundary:**

**o After training, plot the decision boundary to show how the perceptron separates the classes.**

**o Use matplotlib for visualization.**

ANSWER:

CODE:

# Create a meshgrid for visualization by setting the max and min values as -1 and 2 appropriately. as the values are either way between 0 and 1

x\_min, x\_max = -1, 2

y\_min, y\_max = -1, 2

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

# Flatten the meshgrid arrays of each x and y into a 1D array to pass them to the perceptron

xx\_flat = xx.ravel()

yy\_flat = yy.ravel()

# Stack the flattened meshgrid into a 2D array of input points that is coloumns will be of the x and rows will be y and they are transposed

grid\_points = np.vstack((xx\_flat, yy\_flat)).T

# AND perceptron

predictions\_and = perceptron\_and.predict(grid\_points)

predictions\_and = predictions\_and.reshape(xx.shape) # Reshaping the predictions to match the meshgrid

# OR perceptron

predictions\_or = perceptron\_or.predict(grid\_points)

predictions\_or = predictions\_or.reshape(xx.shape)

# decision boundary

plt.figure(figsize=(12, 6))

# Subplot 1: AND gate

plt.subplot(1, 2, 1)

plt.contourf(xx, yy, predictions\_and, alpha=0.8, cmap='coolwarm')

plt.colorbar()

plt.scatter(x\_vector[:, 0], x\_vector[:, 1], c=y\_and, s=100, edgecolors='k', cmap='coolwarm')

plt.title("AND Gate Decision Boundary")

plt.xlabel("X1")

plt.ylabel("X2")

# Subplot 2: OR gate

plt.subplot(1, 2, 2)

plt.contourf(xx, yy, predictions\_or, alpha=0.8, cmap='coolwarm')

plt.colorbar()

plt.scatter(x\_vector[:, 0], x\_vector[:, 1], c=y\_or, s=100, edgecolors='k', cmap='coolwarm')

plt.title("OR Gate Decision Boundary")

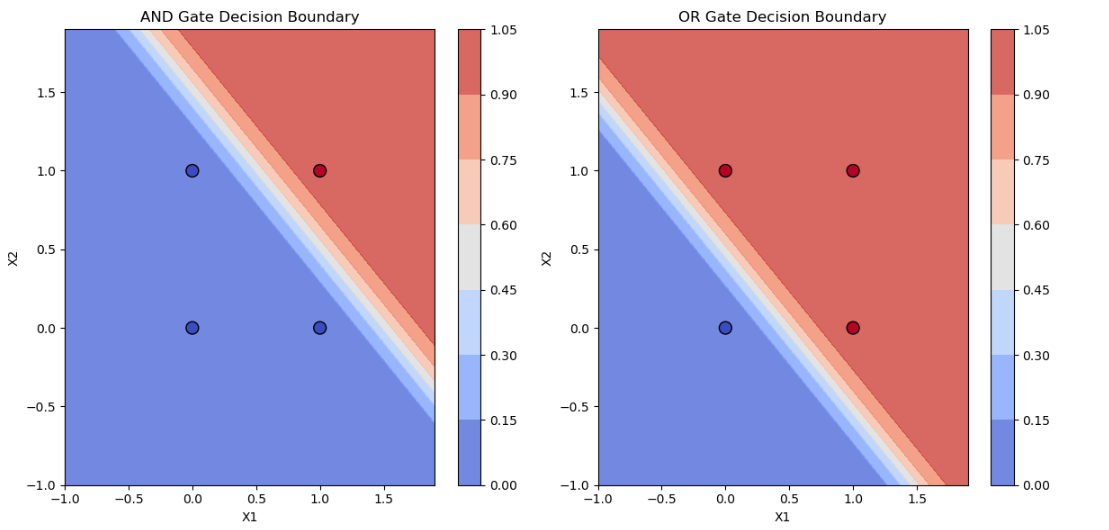
plt.xlabel("X1")

plt.ylabel("X2")

plt.tight\_layout()

plt.show()

OUTPUT:



**3. Extension:**

**o implementing a perceptron for the XOR gate.**

**o Show a single-layer perceptron fails for XOR.**

**o Introduce multi-layer perceptrons (MLP) to overcome**

ANSWER:

CODE:

# XOR Gate data

y\_xor = np.array([0, 1, 1, 0])

perceptron\_xor = Perceptron(input\_size)

perceptron\_xor.train(x\_vector, y\_xor, epochs)

x\_min, x\_max = -1, 2

y\_min, y\_max = -1, 2

xx, yy = np.meshgrid(np.arange(x\_min, x\_max, 0.1), np.arange(y\_min, y\_max, 0.1))

xx\_flat = xx.ravel()

yy\_flat = yy.ravel()

grid\_points = np.vstack((xx\_flat, yy\_flat)).T

predictions\_xor = perceptron\_xor.predict(grid\_points)

predictions\_xor = predictions\_xor.reshape(xx.shape)

plt.figure(figsize=(6, 6))

plt.contourf(xx, yy, predictions\_xor, alpha=0.8, cmap='coolwarm')

plt.colorbar()

plt.scatter(x\_vector[:, 0], x\_vector[:, 1], c=y\_xor, s=100, edgecolors='k', cmap='coolwarm')

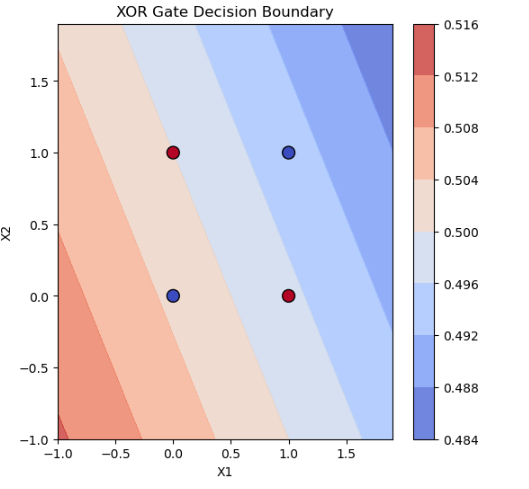
plt.title("XOR Gate Decision Boundary")

plt.xlabel("X1")

plt.ylabel("X2")

plt.show()

OUTPUT:



CODE:

y\_xor = np.array([0, 1, 1, 0])

# MLP Training function

def train\_mlp(x\_vector, y\_xor, hidden\_neurons=2, epochs=10000, lr=0.1):

input\_neurons = x\_vector.shape[1]

output\_neurons = 1

# Initialize weights and biases for the hidden and output layers

w1 = np.random.rand(input\_neurons, hidden\_neurons)

b1 = np.random.rand(hidden\_neurons)

w2 = np.random.rand(hidden\_neurons, output\_neurons)

b2 = np.random.rand(output\_neurons)

# Training loop

for \_ in range(epochs):

# Forward propagation

hidden\_input = np.dot(x\_vector, w1) + b1 # Input to hidden layer

hidden\_output = sigmoid(hidden\_input) # Output from hidden layer

final\_input = np.dot(hidden\_output, w2) + b2 # Input to output layer

final\_output = sigmoid(final\_input) # Output from the networK

# Backpropagation

error = y\_xor.reshape(-1, 1) - final\_output # Calculate the error

d\_output = error \* gradient\_descent(final\_output) # Derivative of output layer

d\_hidden = d\_output.dot(w2.T) \* gradient\_descent(hidden\_output) # Derivative of hidden layer

# Update weights and biases using gradient descent

w2 += hidden\_output.T.dot(d\_output) \* lr # Update weights for output layer

b2 += np.sum(d\_output, axis=0) \* lr # Update bias for output layer

w1 += x\_vector.T.dot(d\_hidden) \* lr # Update weights for hidden layer

b1 += np.sum(d\_hidden, axis=0) \* lr # Update bias for hidden layer

return w1, b1, w2, b2

# Train the MLP on XOR data

w1, b1, w2, b2 = train\_mlp(x\_vector, y\_xor)

# MLP Prediction function

def mlp\_predict(x\_vector):

hidden\_input = np.dot(x\_vector, w1) + b1

hidden\_output = sigmoid(hidden\_input)

final\_input = np.dot(hidden\_output, w2) + b2

return sigmoid(final\_input)

# Visualize the decision boundary for XOR

xx, yy = np.meshgrid(np.linspace(-0.5, 1.5, 100), np.linspace(-0.5, 1.5, 100)) # Create meshgrid

Z = mlp\_predict(np.c\_[xx.ravel(), yy.ravel()]).reshape(xx.shape) # Get predictions for each point

plt.contourf(xx, yy, Z, alpha=0.5) # Plot the decision boundary

plt.scatter(x\_vector[:, 0], x\_vector[:, 1], c=y\_xor, edgecolors='k', marker='o') # Plot the training points

plt.title("Decision Boundary for XOR ")

plt.xlabel("X1")

plt.ylabel("X2")

plt.show()

OUTPUT:

