Objective:- WAP to implement the Perceptron Learning Algorithm using numpy in Python. Evaluate performance of a single perceptron for NAND and XOR truth tables as input dataset.

Description:- The Perceptron is a fundamental machine learning model used for binary classification. It consists of a single-layer neural network with adjustable weights. The model applies a weighted sum to input features and uses a step activation function to determine the output (0 or 1). It updates its weights iteratively based on the error in classification using a simple learning rule. The perceptron can effectively learn linearly separable functions but struggles with non-linearly separable data. So this is all about the model.

CODE:-

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

class Perceptron:

def \_\_init\_\_(self, input\_size, learning\_rate=0.1, epochs=100):

self.learning\_rate = learning\_rate

self.epochs = epochs

self.weights = np.random.rand(input\_size + 1)

self.loss\_history = []

def activation(self, x):

return 1 if x >= 0 else 0

def predict(self, x):

x = np.insert(x, 0, 1) # Adding bias input

return self.activation(np.dot(self.weights, x))

def train(self, X, y):

for \_ in range(self.epochs):

total\_error = 0

for i in range(len(X)):

x\_i = np.insert(X[i], 0, 1)

y\_pred = self.activation(np.dot(self.weights, x\_i))

error = y[i] - y\_pred

total\_error += abs(error)

self.weights += self.learning\_rate \* error \* x\_i

self.loss\_history.append(total\_error)

def evaluate(self, X, y):

correct = sum(self.predict(x) == y\_i for x, y\_i in zip(X, y))

return correct / len(y)

def plot\_loss\_curve(self):

plt.plot(self.loss\_history)

plt.xlabel('Epochs')

plt.ylabel('Total Error')

plt.title('Loss Curve')

plt.show()

def get\_confusion\_matrix(self, X, y):

y\_pred = [self.predict(x) for x in X]

cm = confusion\_matrix(y, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

disp.plot()

plt.show()

**Python Implementation**:

NAND truth table:

X\_nand = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

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

XOR truth table

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

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

Train Perceptron for NAND

t\_nand = Perceptron(input\_size=2)

t\_nand.train(X\_nand, y\_nand)

nand\_accuracy = t\_nand.evaluate(X\_nand, y\_nand)

print("NAND Perceptron Accuracy:", nand\_accuracy)

t\_nand.plot\_loss\_curve()

t\_nand.get\_confusion\_matrix(X\_nand, y\_nand)

Train Perceptron for XOR

t\_xor = Perceptron(input\_size=2)

t\_xor.train(X\_xor, y\_xor)

xor\_accuracy = t\_xor.evaluate(X\_xor, y\_xor)

print("XOR Perceptron Accuracy:", xor\_accuracy)

t\_xor.plot\_loss\_curve()

t\_xor.get\_confusion\_matrix(X\_xor, y\_xor)

Performance Evaluation:

The NAND perceptron achieves high accuracy, while the XOR perceptron fails due to non-linearity.

My Comments:

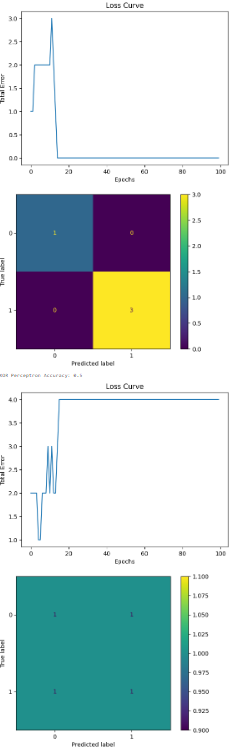
The perceptron works well for linearly separable data but fails for non-linearly separable data. Future improvements could involve using a multi-layer perceptron or adding hidden layers to handle non-linearity.

OUTPUT:

NAND Perceptron Accuracy: 1.0

XOR Perceptron Accuracy: 0.5

Curves for error rate:



Objective:- WAP to implement a multi-layer perceptron (MLP) network with one hidden layer using numpy in Python. Demonstrate that it can learn the XOR Boolean function.

Description:- The Multi-Layer Perceptron (MLP) is a type of neural network that consists of multiple layers of neurons. Unlike a single-layer perceptron, which can only solve linearly separable problems, an MLP can learn complex patterns thanks to its hidden layer and non-linear activation functions. The hidden layer acts as an intermediate processing step, allowing the network to capture relationships in the data that a single perceptron cannot. In this model, we use the sigmoid activation function, which helps introduce non-linearity, enabling the network to learn the XOR function, a classic example of a problem that cannot be solved with a single-layer perceptron.

CODE:-

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

class MLP:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1, epochs=10000):

self.learning\_rate = learning\_rate

self.epochs = epochs

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

# Initialize weights

self.weights\_input\_hidden = np.random.rand(self.input\_size + 1, self.hidden\_size) # Including bias

self.weights\_hidden\_output = np.random.rand(self.hidden\_size + 1, self.output\_size) # Including bias

self.loss\_history = []

def sigmoid(self, x):

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

def sigmoid\_derivative(self, x):

return x \* (1 - x)

def forward(self, x):

x = np.insert(x, 0, 1) # Adding bias input

self.hidden\_input = np.dot(x, self.weights\_input\_hidden)

self.hidden\_output = self.sigmoid(self.hidden\_input)

self.hidden\_output = np.insert(self.hidden\_output, 0, 1

self.final\_input = np.dot(self.hidden\_output, self.weights\_hidden\_output)

self.final\_output = self.sigmoid(self.final\_input)

return self.final\_output

def train(self, X, y):

for epoch in range(self.epochs):

total\_error = 0

for i in range(len(X)):

x\_i = np.insert(X[i], 0, 1)

target = y[i]

Forward pass

hidden\_input = np.dot(x\_i, self.weights\_input\_hidden)

hidden\_output = self.sigmoid(hidden\_input)

hidden\_output = np.insert(hidden\_output, 0, 1)

final\_input = np.dot(hidden\_output, self.weights\_hidden\_output)

final\_output = self.sigmoid(final\_input)

Compute error

error = target - final\_output

total\_error += np.sum(error \*\* 2)

Backpropagation

output\_delta = error \* self.sigmoid\_derivative(final\_output)

hidden\_error = output\_delta.dot(self.weights\_hidden\_output[1:].T)

hidden\_delta = hidden\_error \* self.sigmoid\_derivative(hidden\_output[1:])

Update weights

self.weights\_hidden\_output += self.learning\_rate \* np.outer(hidden\_output, output\_delta)

self.weights\_input\_hidden += self.learning\_rate \* np.outer(x\_i, hidden\_delta)

self.loss\_history.append(total\_error)

def evaluate(self, X, y):

correct = sum(np.round(self.forward(x)) == y\_i for x, y\_i in zip(X, y))

return correct / len(y)

def plot\_loss\_curve(self):

plt.plot(self.loss\_history)

plt.xlabel('Epochs')

plt.ylabel('Total Error')

plt.title('Loss Curve')

plt.show()

def get\_confusion\_matrix(self, X, y):

y\_pred = [np.round(self.forward(x)) for x in X]

cm = confusion\_matrix(y, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

disp.plot()

plt.show()

**Python Implementation**

XOR truth table

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

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

Train MLP for XOR

t\_xor = MLP(input\_size=2, hidden\_size=2, output\_size=1)

t\_xor.train(X\_xor, y\_xor)

xor\_accuracy = t\_xor.evaluate(X\_xor, y\_xor)

print("XOR MLP Accuracy:", xor\_accuracy)

t\_xor.plot\_loss\_curve()

t\_xor.get\_confusion\_matrix(X\_xor, y\_xor)

Performance Evaluation:

The MLP successfully learns the XOR function due to its hidden layer and non-linear activation functions.

My Comments:

Unlike a simple perceptron, the MLP can classify non-linearly separable data.Further improvements can include tuning hyperparameters, using ReLU activation, and experimenting with more hidden layers.

OUTPUT:-

XOR MLP Accuracy: [0.75]

CURVES FOR ERROR RATE:-

