ML LAB-6

K RAHUL

AIE 22174

Q1-

import numpy as np

import matplotlib.pyplot as plt

def step\_function(x):

return 1 if x >= 0 else 0

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

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

W = np.array([10, 0.2, -0.75])

alpha = 0.05

max\_epochs = 1000

# Error convergence threshold

error\_threshold = 0.002

# Function to calculate the output of the perceptron

def predict(X, W):

return step\_function(np.dot(X, W[1:]) + W[0])

# Function to calculate sum square error

def sum\_square\_error(y\_true, y\_pred):

return np.sum((y\_true - y\_pred) \*\* 2)

# List to store error values for plotting

error\_values = []

# Perceptron learning algorithm

for epoch in range(max\_epochs):

errors = []

for i in range(len(X)):

y\_pred = predict(X[i], W)

error = y[i] - y\_pred

errors.append(error)

W[1:] += alpha \* error \* X[i]

W[0] += alpha \* error

epoch\_error = sum\_square\_error(y, [predict(x, W) for x in X])

error\_values.append(epoch\_error)

if epoch\_error <= error\_threshold:

print(f"Converged at epoch {epoch+1}")

break

# Plotting epochs against error values

plt.plot(range(1, len(error\_values) + 1), error\_values)

plt.xlabel('Epochs')

plt.ylabel('Sum Square Error')

plt.title('Epochs vs Error')

plt.grid(True)

plt.show()

Q2-

import numpy as np

import matplotlib.pyplot as plt

# Activation functions

def step\_function(x):

return 1 if x >= 0 else 0

def bipolar\_step\_function(x):

return 1 if x >= 0 else -1

def sigmoid\_function(x):

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

def relu\_function(x):

return max(0, x)

# AND gate training data

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

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

# Initial weights

W = np.array([10, 0.2, -0.75])

# Learning rate

alpha = 0.05

# Maximum number of epochs

max\_epochs = 1000

# Error convergence threshold

error\_threshold = 0.002

# Function to calculate the output of the perceptron

def predict(X, W, activation\_function):

return activation\_function(np.dot(X, W[1:]) + W[0])

# Function to calculate sum square error

def sum\_square\_error(y\_true, y\_pred):

return np.sum((y\_true - y\_pred) \*\* 2)

# Perceptron learning algorithm

def perceptron\_learning(X, y, W, activation\_function):

error\_values = []

for epoch in range(max\_epochs):

errors = []

for i in range(len(X)):

y\_pred = predict(X[i], W, activation\_function)

error = y[i] - y\_pred

errors.append(error)

W[1:] += alpha \* error \* X[i]

W[0] += alpha \* error

epoch\_error = sum\_square\_error(y, [predict(x, W, activation\_function) for x in X])

error\_values.append(epoch\_error)

if epoch\_error <= error\_threshold:

print(f"Converged at epoch {epoch+1}")

break

return error\_values

# Perform experiments with different activation functions

activation\_functions = [bipolar\_step\_function, sigmoid\_function, relu\_function]

activation\_labels = ['Bi-Polar Step', 'Sigmoid', 'ReLU']

for i, activation\_function in enumerate(activation\_functions):

print(f"\nExperiment with {activation\_labels[i]} activation function:")

W = np.array([10, 0.2, -0.75]) # Reset initial weights for each experiment

error\_values = perceptron\_learning(X, y, W, activation\_function)

# Plotting epochs against error values

plt.plot(range(1, len(error\_values) + 1), error\_values, label=activation\_labels[i])

plt.xlabel('Epochs')

plt.ylabel('Sum Square Error')

plt.title('Epochs vs Error')

plt.grid(True)

plt.legend()

plt.show()

Q3-

import numpy as np

import matplotlib.pyplot as plt

# Step activation function

def step\_function(x):

return 1 if x >= 0 else 0

# AND gate training data

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

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

# Initial weights

W = np.array([10, 0.2, -0.75])

# Maximum number of epochs

max\_epochs = 1000

# Error convergence threshold

error\_threshold = 0.002

# Function to calculate the output of the perceptron

def predict(X, W):

return step\_function(np.dot(X, W[1:]) + W[0])

# Function to calculate sum square error

def sum\_square\_error(y\_true, y\_pred):

return np.sum((y\_true - y\_pred) \*\* 2)

# Function to perform perceptron learning with given learning rate

def perceptron\_learning(X, y, W, alpha):

for epoch in range(max\_epochs):

for i in range(len(X)):

y\_pred = predict(X[i], W)

error = y[i] - y\_pred

W[1:] += alpha \* error \* X[i]

W[0] += alpha \* error

epoch\_error = sum\_square\_error(y, [predict(x, W) for x in X])

if epoch\_error <= error\_threshold:

return epoch + 1 # Return the number of iterations taken to converge

return max\_epochs # If not converged, return maximum number of epochs

# List of learning rates to test

learning\_rates = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]

# List to store the number of iterations taken for each learning rate

iterations = []

# Perform experiments with different learning rates

for alpha in learning\_rates:

W = np.array([10, 0.2, -0.75]) # Reset initial weights for each experiment

iterations.append(perceptron\_learning(X, y, W, alpha))

# Plotting learning rates against the number of iterations taken for learning to converge

plt.plot(learning\_rates, iterations, marker='o')

plt.xlabel('Learning Rate')

plt.ylabel('Iterations to Converge')

plt.title('Learning Rate vs Iterations to Converge')

plt.grid(True)

plt.show()

Q-5

import numpy as np

# Customer data

data = np.array([

[20,6,2,386,1], # C\_1

[16,3,6,289,1], # C\_2

[27,6,2,393, 1], # C\_3

[19,1,2,110, 0], # C\_4

[24,4,2,280, 1], # C\_5

[22,1,5,167, 0], # C\_6

[15,4,2,271, 1], # C\_7

[18,4,2,274, 1], # C\_8

[21,1,4,148, 0], # C\_9

[16,2,4,198, 0] # C\_10

])

# Sigmoid activation function

def sigmoid(x):

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

# Initialize weights and bias

np.random.seed(0) # for reproducibility

weights = np.random.randn(5)

learning\_rate = 0.01

epochs = 1000

# Extract features and labels

X = data[:, :-1] # Features

y = data[:, -1] # Labels

# Training the perceptron

for epoch in range(epochs):

# Forward pass

weighted\_sum = np.dot(X, weights[1:]) + weights[0]

predictions = sigmoid(weighted\_sum)

# Backpropagation

error = y - predictions

adjustments = learning\_rate \* np.dot(X.T, error \* predictions \* (1 - predictions))

# Update weights

weights[1:] += adjustments

weights[0] += np.sum(adjustments)

# Classify transactions

# Classify transactions

def classify\_transaction(customer):

features = np.array(customer)

weighted\_sum = np.dot(features, weights[1:]) + weights[0] # Include bias

prediction = sigmoid(weighted\_sum)

if prediction >= 0.5: # Use 0.5 as the threshold

return "High Value"

else:

return "Low Value"

# Test the perceptron

test\_customers = [

[18, 5,4, 200], # Test a high value transaction

[10, 1,2, 100] # Test a low value transaction

]

for customer in test\_customers:

prediction = classify\_transaction(customer)

print(f"Customer: {customer}, Predicted Probability: {prediction}")

for customer in test\_customers:

print(f"Customer: {customer}, Predicted Class: {classify\_transaction(customer)}")