Lab 6

AIM:

Write a program to implement perceptron neural network for Classification/Regression.

DESCRIPTION:

A perceptron is a fundamental building block of a neural network. It models a neuron with weighted inputs, an activation function, and a single output. For classification tasks like MNIST digit recognition, the perceptron uses an activation function (e.g., ReLU, softmax). For regression tasks like housing price prediction, the perceptron can use linear activation functions.

This program demonstrates two tasks:

1. Classification: Recognizing handwritten digits using the MNIST dataset.
2. Regression: Predicting housing prices using the Boston Housing dataset.

PROGRAM:

Classification of Handwritten digits using MNIST dataset and Regression of prices using Boston Housing dataset:

import numpy as np import tensorflow as tf from tensorflow.keras.models import Sequential from tensorflow.keras.layers import Dense, Flatten from tensorflow.keras.datasets import mnist, boston\_housing

def mnist\_classification():

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data() x\_train, x\_test = x\_train / 255.0, x\_test / 255.0

model = Sequential([Flatten(input\_shape=(28, 28)), Dense(128, activation='relu'), Dense(10, activation='softmax')])

model.compile(optimizer="adam", loss="sparse\_categorical\_crossentropy", metrics=["accuracy"]) model.fit(x\_train, y\_train, epochs=5, validation\_data=(x\_test, y\_test)) test\_loss, test\_acc = model.evaluate(x\_test, y\_test) print("Classification Test Accuracy:", test\_acc)

def boston\_regression():

(x\_train, y\_train), (x\_test, y\_test) = boston\_housing.load\_data() mean = x\_train.mean(axis=0) std = x\_train.std(axis=0) x\_train = (x\_train - mean) / std

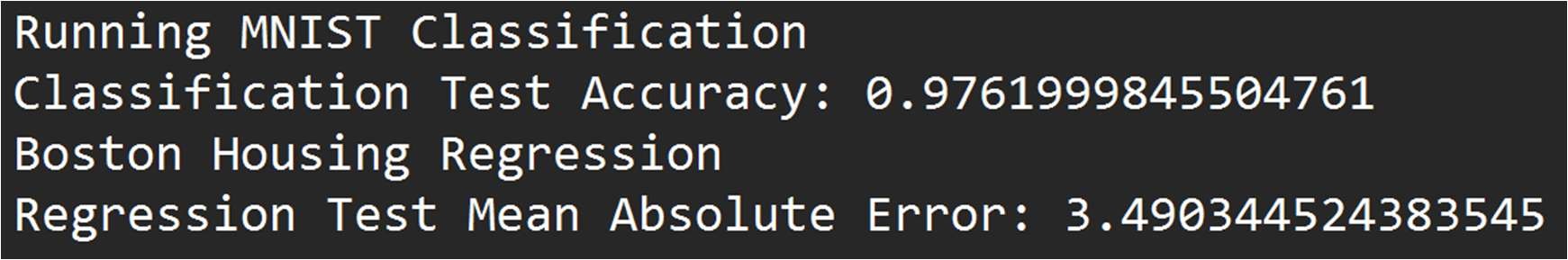
x\_test = (x\_test - mean) / std

model = Sequential([Dense(64, activation='relu', input\_shape=(x\_train.shape[1],)), Dense(64, activation='relu'), Dense(1)]) model.compile(optimizer="adam", loss="mse", metrics=["mae"]) model.fit(x\_train, y\_train, epochs=50, validation\_split=0.2, verbose=1) test\_loss, test\_mae = model.evaluate(x\_test, y\_test) print("Regression Test Mean Absolute Error:", test\_mae)

if \_\_name\_\_ == "\_\_main\_\_":

print("Running MNIST Classification") mnist\_classification() print("Boston Housing Regression") boston\_regression()

OUTPUT:



Lab 7

AIM:

Write a program to implement Backpropagation in neural network.

DESCRIPTION:

Backpropagation (short for "backward propagation of errors") is a supervised learning algorithm used in training artificial neural networks. It calculates the gradient of the loss function with respect to each weight in the network, iteratively updating the weights using techniques like gradient descent. The steps include:

1. Forward Propagation: Compute outputs using input data and current weights.
2. Error Calculation: Measure the difference between the predicted and actual outputs (using a loss function like MSE or cross-entropy).
3. Backward Propagation: Calculate gradients of loss with respect to weights using the chain rule.
4. Weight Update: Adjust weights to minimize the loss function.

PROGRAM:

import numpy as np

def sigmoid(x): return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x): return x \* (1 - x) class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1): self.weights\_input\_hidden = np.random.randn(input\_size, hidden\_size) self.weights\_hidden\_output = np.random.randn(hidden\_size, output\_size) self.bias\_hidden = np.random.randn(1, hidden\_size) self.bias\_output = np.random.randn(1, output\_size) self.learning\_rate = learning\_rate def forward(self, X):

self.input = X

self.hidden\_layer\_input = np.dot(self.input, self.weights\_input\_hidden) + self.bias\_hidden self.hidden\_layer\_output = sigmoid(self.hidden\_layer\_input) self.output\_layer\_input = np.dot(self.hidden\_layer\_output, self.weights\_hidden\_output) + self.bias\_output

self.output = sigmoid(self.output\_layer\_input) return self.output

def backward(self, y\_true): error = y\_true - self.output output\_gradient = error \* sigmoid\_derivative(self.output) hidden\_error = np.dot(output\_gradient, self.weights\_hidden\_output.T) hidden\_gradient = hidden\_error \* sigmoid\_derivative(self.hidden\_layer\_output) self.weights\_hidden\_output += np.dot(self.hidden\_layer\_output.T, output\_gradient) \* self.learning\_rate self.bias\_output += np.sum(output\_gradient, axis=0, keepdims=True) \* self.learning\_rate self.weights\_input\_hidden += np.dot(self.input.T, hidden\_gradient) \* self.learning\_rate self.bias\_hidden += np.sum(hidden\_gradient, axis=0, keepdims=True) \* self.learning\_rate

def train(self, X, y, epochs=10000):

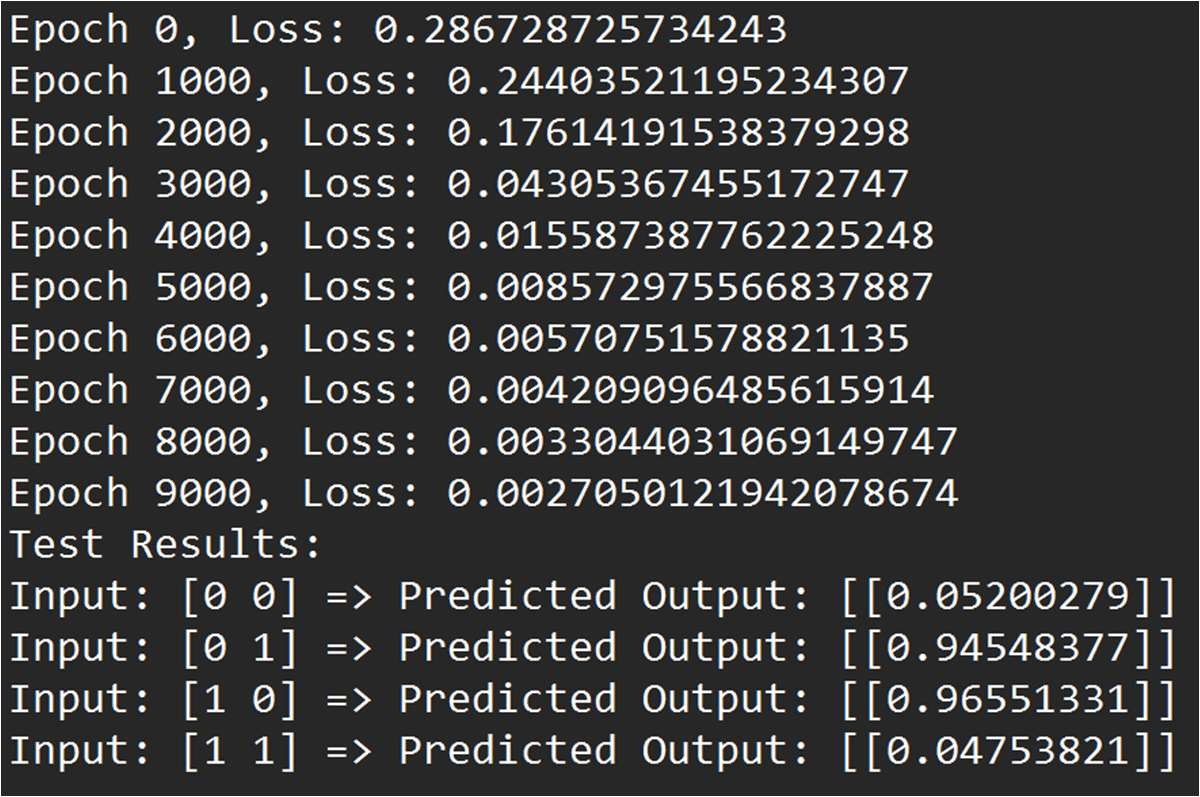
for epoch in range(epochs):

self.forward(X) self.backward(y) if epoch % 1000 == 0:

loss = np.mean((y - self.output) \*\* 2) print(f"Epoch {epoch}, Loss: {loss}")

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input for XOR y = np.array([[0], [1], [1], [0]]) # Target output for XOR nn = NeuralNetwork(input\_size=2, hidden\_size=4, output\_size=1, learning\_rate=0.1) nn.train(X, y, epochs=10000) print("Test Results:") for i in range(len(X)): print(f"Input: {X[i]} => Predicted Output: {nn.forward(X[i])}")

OUTPUT:



Lab 8

Aim:

Write a program to implement Backpropagation neural network for Classification/Regression.

DESCRIPTION:

Backpropagation is a supervised learning algorithm used for training artificial neural networks. It is based on the chain rule of calculus and allows the model to update the weights in a way that reduces the error (loss). The process involves two main steps:

1. Forward pass: Input is passed through the network, and output is computed.
2. Backward pass: The error is propagated backward through the network to update the weights using the gradient descent algorithm.

In this program, we implement a simple feedforward neural network for classification. The network has:

An input layer (2 neurons, assuming 2-dimensional input).

A hidden layer (with configurable size).

An output layer (1 neuron for binary classification).

The goal is to classify the points using backpropagation.

PROGRAM:

import numpy as np

def sigmoid(x): return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x): return x \* (1 - x)

class NeuralNetwork: def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1): self.weights\_input\_hidden = np.random.randn(input\_size, hidden\_size) self.weights\_hidden\_output = np.random.randn(hidden\_size, output\_size) self.bias\_hidden = np.random.randn(1, hidden\_size) self.bias\_output = np.random.randn(1, output\_size) self.learning\_rate = learning\_rate

def forward(self, X):

self.input = X

self.hidden\_layer\_input = np.dot(self.input, self.weights\_input\_hidden) + self.bias\_hidden

self.hidden\_layer\_output = sigmoid(self.hidden\_layer\_input) self.output\_layer\_input = np.dot(self.hidden\_layer\_output, self.weights\_hidden\_output)

+ self.bias\_output self.output = sigmoid(self.output\_layer\_input) return self.output

def backward(self, y\_true): error = y\_true - self.output output\_gradient = error \* sigmoid\_derivative(self.output) hidden\_error = np.dot(output\_gradient, self.weights\_hidden\_output.T) hidden\_gradient = hidden\_error \* sigmoid\_derivative(self.hidden\_layer\_output) self.weights\_hidden\_output += np.dot(self.hidden\_layer\_output.T, output\_gradient) \* self.learning\_rate self.bias\_output += np.sum(output\_gradient, axis=0, keepdims=True) \* self.learning\_rate self.weights\_input\_hidden += np.dot(self.input.T, hidden\_gradient) \* self.learning\_rate self.bias\_hidden += np.sum(hidden\_gradient, axis=0, keepdims=True) \* self.learning\_rate

def train(self, X, y, epochs=10000):

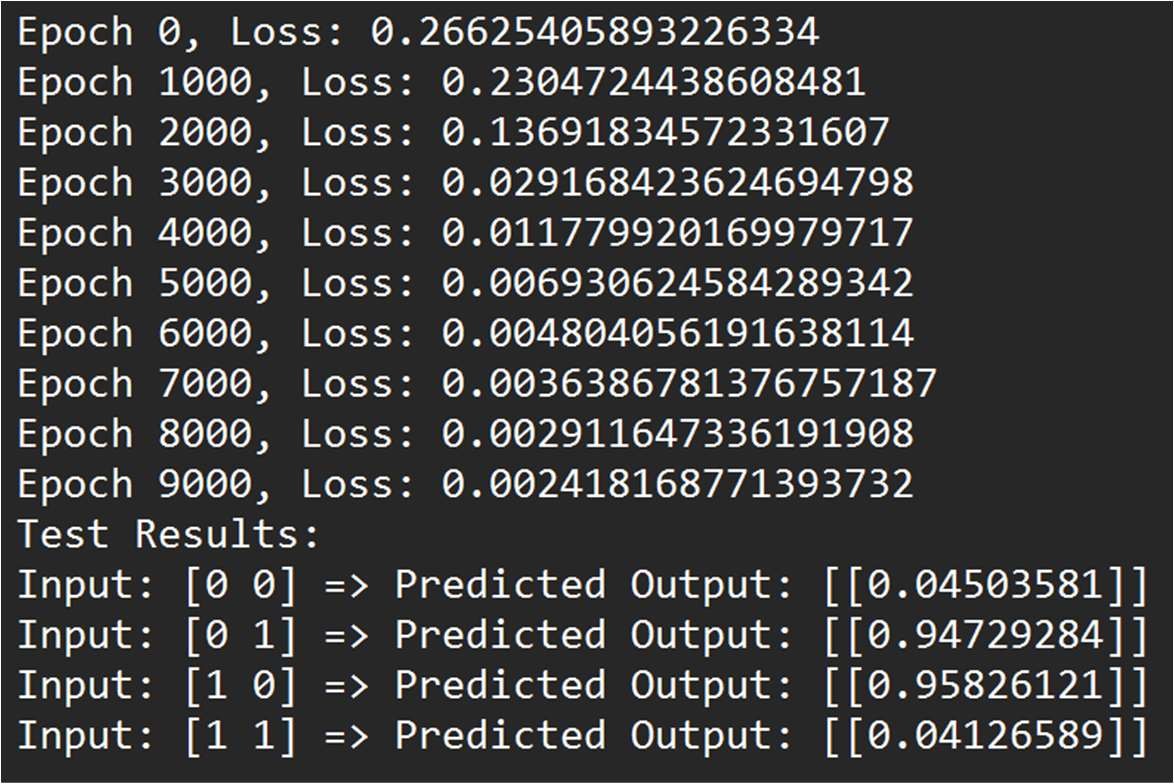
for epoch in range(epochs):

self.forward(X) self.backward(y) if epoch % 1000 == 0:

loss = np.mean((y - self.output) \*\* 2) print(f"Epoch {epoch}, Loss: {loss}")

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input for XOR y = np.array([[0], [1], [1], [0]]) # Target output for XOR nn = NeuralNetwork(input\_size=2, hidden\_size=4, output\_size=1, learning\_rate=0.1) nn.train(X, y, epochs=10000) print("Test Results:") for i in range(len(X)): print(f"Input: {X[i]} => Predicted Output: {nn.forward(X[i])}")

OUTPUT:



## Lab 11

AIM:

Implement Fuzzy operations Union, Intersections, Complement.

DESCRIPTION:

In fuzzy logic, sets are represented by membership functions that assign a degree of membership to each element in the set. These degrees range from 0 (no membership) to 1 (full membership). Fuzzy set operations are extensions of classical set operations, such as:

Union: The union of two fuzzy sets returns the maximum membership value for each element from the two sets. μₐ∪ᵦ(x) = max(μₐ(x), μᵦ(x))

Intersection: The intersection of two fuzzy sets returns the minimum membership value for each element from the two sets. μₐ∩ᵦ(x) = min(μₐ(x), μᵦ(x))

Complement: The complement of a fuzzy set inverts the membership values, so each element's membership value becomes 1 - μ(x). μₐʳ(x) = 1 - μₐ(x)

We will implement these operations on fuzzy sets where each fuzzy set is represented by a list of tuples. Each tuple contains an element and its associated membership value.

PROGRAM:

def fuzzy\_union(set\_A, set\_B):

union\_result = [] for (x, mu\_A) in set\_A:

mu\_B = next((mu\_B for (y, mu\_B) in set\_B if x == y), 0) union\_result.append((x, max(mu\_A, mu\_B))) return union\_result

def fuzzy\_intersection(set\_A, set\_B):

intersection\_result = [] for (x, mu\_A) in set\_A:

mu\_B = next((mu\_B for (y, mu\_B) in set\_B if x == y), 0) intersection\_result.append((x, min(mu\_A, mu\_B))) return intersection\_result

def fuzzy\_complement(set\_A):

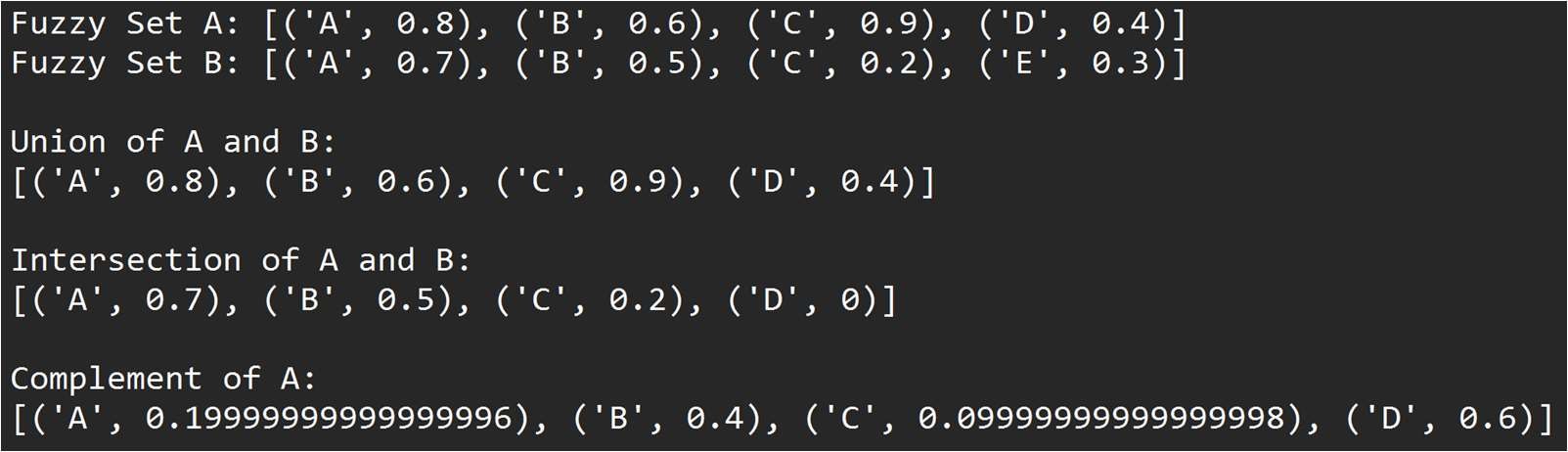
complement\_result = [] for (x, mu\_A) in set\_A: complement\_result.append((x, 1 - mu\_A)) return complement\_result

1. = [('A', 0.8), ('B', 0.6), ('C', 0.9), ('D', 0.4)]
2. = [('A', 0.7), ('B', 0.5), ('C', 0.2), ('E', 0.3)]

union\_result = fuzzy\_union(A, B) intersection\_result = fuzzy\_intersection(A, B) complement\_result\_A = fuzzy\_complement(A)

print("Fuzzy Set A:", A) print("Fuzzy Set B:", B) print("\nUnion of A and B:") print(union\_result) print("\nIntersection of A and B:") print(intersection\_result) print("\nComplement of A:") print(complement\_result\_A)

OUTPUT:



## Lab 12

Write a program to implement the concept of Genetic Algorithm.

DESCRIPTION:

A Genetic Algorithm (GA) mimics the process of natural selection. It maintains a population of possible solutions to a problem, evaluates their fitness, and then uses genetic operations like selection, crossover, and mutation to evolve better solutions over successive generations.

In this case, the target is to evolve a population of strings towards a given target string.

The basic steps are:

Initialization: Randomly create an initial population of strings.

Selection: Select individuals from the population based on their fitness (how close they are to the target string).

Crossover: Combine pairs of individuals to create new offspring.

Mutation: Apply random changes to the offspring to maintain genetic diversity. Termination: Stop the algorithm when a solution matches the target string.

PROGRAM:

import random

TARGET = "Hello, Genetic Algorithm!"

POPULATION\_SIZE = 100

MUTATION\_RATE = 0.01 GENERATION\_LIMIT = 1000

def fitness(individual):

return sum(1 for i, char in enumerate(individual) if char == TARGET[i])

def create\_individual(): return

''.join(random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY

Z, !') for \_ in range(len(TARGET)))

def select(population):

total\_fitness = sum(fitness(individual) for individual in population) selection\_probs = [fitness(individual) / total\_fitness for individual in population] selected = random.choices(population, weights=selection\_probs, k=2) return selected

def crossover(parent1, parent2):

crossover\_point = random.randint(0, len(parent1)-1)

child1 = parent1[:crossover\_point] + parent2[crossover\_point:] child2 = parent2[:crossover\_point] + parent1[crossover\_point:] return child1, child2

def mutate(individual):

individual = list(individual) for i in range(len(individual)): if random.random() < MUTATION\_RATE:

individual[i] =

random.choice('abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ, !') return ''.join(individual)

def genetic\_algorithm():

population = [create\_individual() for \_ in range(POPULATION\_SIZE)] generation = 0 while generation < GENERATION\_LIMIT:

population.sort(key=fitness, reverse=True) if fitness(population[0]) == len(TARGET):

print(f"Target reached in generation {generation}: {population[0]}") break

next\_generation = population[:POPULATION\_SIZE//2] while len(next\_generation) < POPULATION\_SIZE:

parent1, parent2 = select(population) child1, child2 = crossover(parent1, parent2) next\_generation.append(mutate(child1)) next\_generation.append(mutate(child2)) population = next\_generation generation += 1 else:

print("Target not reached within generation limit.") genetic\_algorithm()

OUTPUT:

