Prac9

import numpy as np

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

class BackpropagationNN:

def \_\_init\_\_(self, input\_nodes, hidden\_nodes, output\_nodes, learning\_rate):

self.input\_nodes = input\_nodes

self.hidden\_nodes = hidden\_nodes

self.output\_nodes = output\_nodes

self.learning\_rate = learning\_rate

self.weights\_input\_hidden = np.random.rand(self.input\_nodes, self.hidden\_nodes) - 0.5

self.weights\_hidden\_output = np.random.rand(self.hidden\_nodes, self.output\_nodes) - 0.5

self.bias\_hidden = np.random.rand(1, self.hidden\_nodes) - 0.5

self.bias\_output = np.random.rand(1, self.output\_nodes) - 0.5

def forward\_propagate(self, X):

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

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

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

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

return self.final\_output

def backpropagate(self, X, y):

output\_error = y - self.final\_output

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

hidden\_error = np.dot(output\_delta, self.weights\_hidden\_output.T)

hidden\_delta = hidden\_error \* sigmoid\_derivative(self.hidden\_output)

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

self.bias\_output += self.learning\_rate \* np.sum(output\_delta, axis=0, keepdims=True)

self.weights\_input\_hidden += self.learning\_rate \* np.dot(X.T, hidden\_delta)

self.bias\_hidden += self.learning\_rate \* np.sum(hidden\_delta, axis=0, keepdims=True)

# XOR Training Data

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

[0, 1],

[1, 0],

[1, 1]])

y = np.array([[0],

[1],

[1],

[0]])

# Create and Train Neural Network

nn = BackpropagationNN(input\_nodes=2, hidden\_nodes=4, output\_nodes=1, learning\_rate=1)

print("\nTRAINING PROCESS:")

reached\_exact\_output = False

iteration = 0

while not reached\_exact\_output:

iteration += 1

outputs = nn.forward\_propagate(X)

nn.backpropagate(X, y)

print(f"\nITERATION {iteration}")

reached\_exact\_output = True

for inp, expected, output in zip(X, y, outputs):

difference = expected - output

gradient\_output = sigmoid\_derivative(output)

print(f"INPUT: {inp}, EXPECTED: {expected}, OUTPUT: {output}, GRADIENT OUTPUT: {gradient\_output}")

if not np.isclose(output, expected, atol=0.01):

reached\_exact\_output = False