9.TRAVELLING SALESPERSON

import itertools

# Define a function to calculate the distance between two cities

def calculate\_distance(city1, city2):

return ((city1[0] - city2[0])\*\*2 + (city1[1] - city2[1])\*\*2) \*\* 0.5

# Define a function to calculate the total distance of a path

def total\_path\_distance(path, distances):

return sum(distances[path[i]][path[i + 1]] for i in range(len(path) - 1)) + distances[path[-1]][path[0]]

# Main TSP function using brute-force

def traveling\_salesman\_bruteforce(cities):

num\_cities = len(cities)

# Precompute distances between each pair of cities

distances = [[calculate\_distance(cities[i], cities[j]) for j in range(num\_cities)] for i in range(num\_cities)]

# Generate all possible paths

min\_path = None

min\_distance = float('inf')

for path in itertools.permutations(range(num\_cities)):

distance = total\_path\_distance(path, distances)

if distance < min\_distance:

min\_distance = distance

min\_path = path

return min\_path, min\_distance

# List of cities as (x, y) coordinates

cities = [

(0, 0), # City A

(1, 3), # City B

(4, 3), # City C

(6, 1) # City D

]

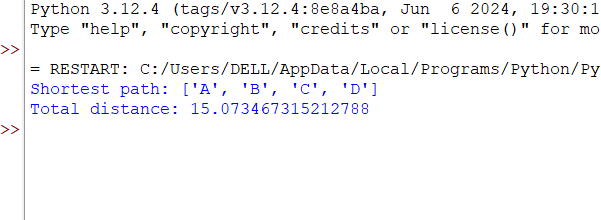
# Solve TSP

path, distance = traveling\_salesman\_bruteforce(cities)

print("Shortest path:", [chr(65 + city) for city in path]) # Convert city index to letter

print("Total distance:", distance)

OUTPUT



10. A\*ALGORITHM

import heapq

class Node:

def \_\_init\_\_(self, x, y, cost=0, heuristic=0):

self.x = x # X-coordinate

self.y = y # Y-coordinate

self.cost = cost # G(n): Cost from start to current node

self.heuristic = heuristic # H(n): Estimated cost to goal

self.f = cost + heuristic # F(n) = G(n) + H(n)

self.parent = None # To reconstruct the path

def \_\_lt\_\_(self, other):

return self.f < other.f

def heuristic(a, b):

# Using Manhattan distance as the heuristic

return abs(a.x - b.x) + abs(a.y - b.y)

def a\_star(grid, start, goal):

# Define open and closed sets

open\_set = []

closed\_set = set()

# Initialize start node

start\_node = Node(start[0], start[1])

goal\_node = Node(goal[0], goal[1])

start\_node.heuristic = heuristic(start\_node, goal\_node)

start\_node.f = start\_node.heuristic

# Push the start node into the priority queue

heapq.heappush(open\_set, start\_node)

# Possible moves (right, left, up, down)

directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

while open\_set:

current\_node = heapq.heappop(open\_set)

# If we reached the goal, reconstruct the path

if (current\_node.x, current\_node.y) == (goal\_node.x, goal\_node.y):

path = []

while current\_node:

path.append((current\_node.x, current\_node.y))

current\_node = current\_node.parent

return path[::-1] # Return reversed path from start to goal

closed\_set.add((current\_node.x, current\_node.y))

# Explore neighbors

for direction in directions:

neighbor\_x = current\_node.x + direction[0]

neighbor\_y = current\_node.y + direction[1]

# Check if neighbor is within bounds and walkable

if 0 <= neighbor\_x < len(grid) and 0 <= neighbor\_y < len(grid[0]) and grid[neighbor\_x][neighbor\_y] == 0:

if (neighbor\_x, neighbor\_y) in closed\_set:

continue # Skip if already in closed set

# Calculate cost and heuristic

new\_cost = current\_node.cost + 1 # Assuming uniform cost of 1 per move

neighbor\_node = Node(neighbor\_x, neighbor\_y, new\_cost)

neighbor\_node.heuristic = heuristic(neighbor\_node, goal\_node)

neighbor\_node.f = neighbor\_node.cost + neighbor\_node.heuristic

neighbor\_node.parent = current\_node

# Check if this path to neighbor is better or if it's unvisited

if not any(n.x == neighbor\_node.x and n.y == neighbor\_node.y and n.f <= neighbor\_node.f for n in open\_set):

heapq.heappush(open\_set, neighbor\_node)

return None # Return None if no path is found

# Define the grid (0: open, 1: obstacle)

grid = [

[0, 1, 0, 0, 0],

[0, 1, 0, 1, 0],

[0, 0, 0, 1, 0],

[0, 1, 1, 1, 0],

[0, 0, 0, 0, 0]

]

# Set the start and goal positions

start = (0, 0) # Top-left corner

goal = (4, 4) # Bottom-right corner

# Run A\* algorithm

path = a\_star(grid, start, goal)

# Print the result

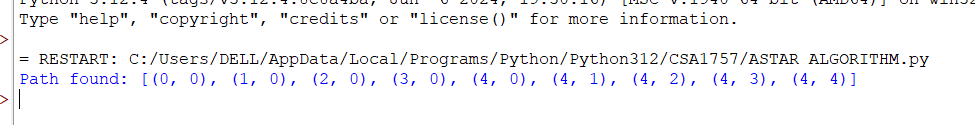
if path:

print("Path found:", path)

else:

print("No path found.")

OUTPUT



11.MAP COLOURING

# Define the map and constraints for adjacent regions

regions = {

"WA": ["NT", "SA"],

"NT": ["WA", "SA", "Q"],

"SA": ["WA", "NT", "Q", "NSW", "V"],

"Q": ["NT", "SA", "NSW"],

"NSW": ["Q", "SA", "V"],

"V": ["SA", "NSW"],

"T": []

}

# Define colors available

colors = ["Red", "Green", "Blue", "Yellow"]

# Function to check if the current color assignment is valid

def is\_valid(region, color, assignment):

for neighbor in regions[region]:

if neighbor in assignment and assignment[neighbor] == color:

return False

return True

# Backtracking function to assign colors

def color\_map(assignment={}):

# If all regions are assigned, return the solution

if len(assignment) == len(regions):

return assignment

# Select an unassigned region

unassigned\_regions = [region for region in regions if region not in assignment]

region = unassigned\_regions[0]

# Try each color for the selected region

for color in colors:

if is\_valid(region, color, assignment):

assignment[region] = color # Assign color

# Recursively assign colors to the rest of the map

result = color\_map(assignment)

if result: # If successful, return the result

return result

# If not successful, backtrack

del assignment[region]

return None # No valid color found, backtrack

# Run the map coloring function and print the result

solution = color\_map()

if solution:

print("Solution found:")

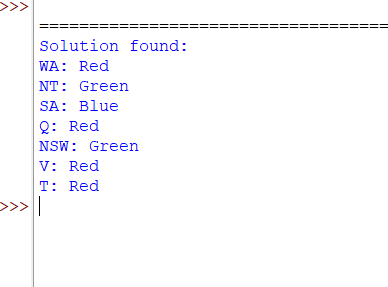
for region, color in solution.items():

print(f"{region}: {color}")

else:

print("No solution found.")

OUTPUT



12.TIC TAC TOE

# Tic Tac Toe Game in Python

# Initialize the board as a list of strings (empty spaces initially)

board = [" " for \_ in range(9)]

# Function to print the current state of the board

def print\_board():

print(" {} | {} | {} ".format(board[0], board[1], board[2]))

print("---+---+---")

print(" {} | {} | {} ".format(board[3], board[4], board[5]))

print("---+---+---")

print(" {} | {} | {} ".format(board[6], board[7], board[8]))

# Function to check if there is a winner

def check\_winner(player):

# All possible winning combinations

win\_conditions = [

[0, 1, 2], [3, 4, 5], [6, 7, 8], # Rows

[0, 3, 6], [1, 4, 7], [2, 5, 8], # Columns

[0, 4, 8], [2, 4, 6] # Diagonals

]

for condition in win\_conditions:

if board[condition[0]] == board[condition[1]] == board[condition[2]] == player:

return True

return False

# Function to check if the board is full

def is\_board\_full():

return " " not in board

# Main function to play the game

def play\_game():

current\_player = "X" # Player X always starts

while True:

print\_board()

print("Player {}'s turn. Enter a position (1-9):".format(current\_player))

try:

move = int(input()) - 1 # Get the position from the player (1-9 to 0-8)

if move < 0 or move >= 9:

print("Invalid position! Choose a position from 1 to 9.")

continue

if board[move] != " ":

print("Position already taken! Choose another.")

continue

except ValueError:

print("Please enter a valid number between 1 and 9.")

continue

# Make the move

board[move] = current\_player

# Check for a win

if check\_winner(current\_player):

print\_board()

print("Player {} wins!".format(current\_player))

break

# Check for a tie

if is\_board\_full():

print\_board()

print("It's a tie!")

break

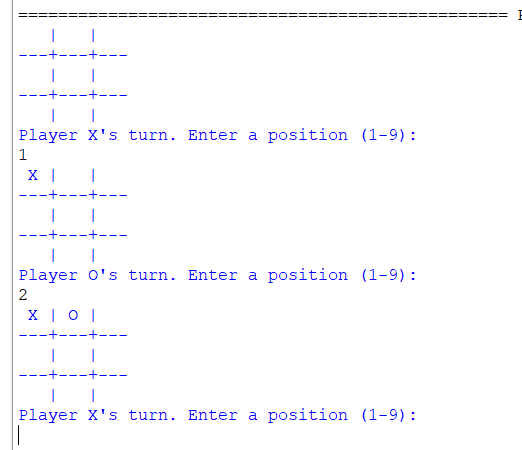
# Switch players

current\_player = "O" if current\_player == "X" else "X"

# Start the game

play\_game()

OUTPUT



13.MIN MAX ALGORITHM

import math

# Define the Connect Four game board

ROWS = 6

COLS = 7

EMPTY = 0

PLAYER\_1 = 1

PLAYER\_2 = -1

# Initialize the board with zeros (EMPTY spots)

def create\_board():

return [[EMPTY for \_ in range(COLS)] for \_ in range(ROWS)]

# Check if the move is valid (column is not full)

def is\_valid\_move(board, col):

return board[0][col] == EMPTY

# Make a move on the board

def make\_move(board, col, player):

for row in range(ROWS-1, -1, -1):

if board[row][col] == EMPTY:

board[row][col] = player

break

# Check if the current player has won the game

def is\_winner(board, player):

# Check horizontal, vertical, and diagonal

for row in range(ROWS):

for col in range(COLS):

if board[row][col] == player:

# Horizontal check

if col + 3 < COLS and all(board[row][col + i] == player for i in range(4)):

return True

# Vertical check

if row + 3 < ROWS and all(board[row + i][col] == player for i in range(4)):

return True

# Diagonal check (\ direction)

if row + 3 < ROWS and col + 3 < COLS and all(board[row + i][col + i] == player for i in range(4)):

return True

# Diagonal check (/ direction)

if row - 3 >= 0 and col + 3 < COLS and all(board[row - i][col + i] == player for i in range(4)):

return True

return False

# Evaluate the board for the Min-Max algorithm

def evaluate(board):

if is\_winner(board, PLAYER\_1):

return 1000

elif is\_winner(board, PLAYER\_2):

return -1000

return 0

# Min-Max Algorithm with Alpha-Beta Pruning

def minimax(board, depth, alpha, beta, maximizing\_player):

valid\_moves = [c for c in range(COLS) if is\_valid\_move(board, c)]

# Base case: If the depth is 0 or the game is over, return the evaluation

if depth == 0 or len(valid\_moves) == 0:

return evaluate(board)

if maximizing\_player:

max\_eval = -math.inf

for move in valid\_moves:

new\_board = [row[:] for row in board]

make\_move(new\_board, move, PLAYER\_1)

eval = minimax(new\_board, depth - 1, alpha, beta, False)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha:

break

return max\_eval

else:

min\_eval = math.inf

for move in valid\_moves:

new\_board = [row[:] for row in board]

make\_move(new\_board, move, PLAYER\_2)

eval = minimax(new\_board, depth - 1, alpha, beta, True)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha:

break

return min\_eval

# Find the best move for the current player (Maximizing Player)

def find\_best\_move(board):

best\_value = -math.inf

best\_move = None

valid\_moves = [c for c in range(COLS) if is\_valid\_move(board, c)]

for move in valid\_moves:

new\_board = [row[:] for row in board]

make\_move(new\_board, move, PLAYER\_1)

move\_value = minimax(new\_board, 5, -math.inf, math.inf, False)

if move\_value > best\_value:

best\_value = move\_value

best\_move = move

return best\_move

# Print the board

def print\_board(board):

for row in board:

print(" | ".join(str(cell) for cell in row))

print("\n")

# Main game loop for Connect Four using Min-Max

def play\_game():

board = create\_board()

game\_over = False

turn = PLAYER\_1 # Player 1 starts

while not game\_over:

print\_board(board)

if turn == PLAYER\_1:

print("Player 1's turn (Maximizing Player)")

move = find\_best\_move(board)

print(f"Player 1 chooses column {move}")

else:

print("Player 2's turn (Minimizing Player)")

move = int(input("Enter column (0-6): "))

if is\_valid\_move(board, move):

make\_move(board, move, turn)

if is\_winner(board, turn):

print\_board(board)

print(f"Player {turn} wins!")

game\_over = True

else:

turn = PLAYER\_2 if turn == PLAYER\_1 else PLAYER\_1

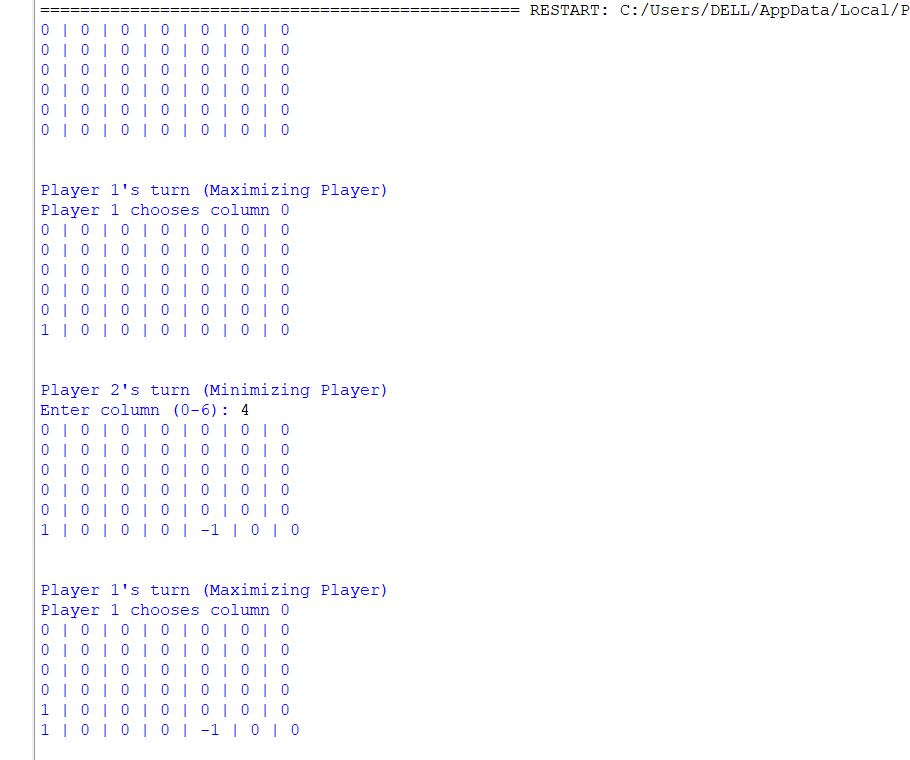
else:

print("Invalid move. Try again.")

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

play\_game()

OUTPUT



14.ALPHA BETA PRUNING

import math

# Constants for Connect Four

ROWS = 6

COLS = 7

EMPTY = 0

PLAYER\_1 = 1 # Maximizing Player

PLAYER\_2 = -1 # Minimizing Player

# Initialize the Connect Four board

def create\_board():

return [[EMPTY for \_ in range(COLS)] for \_ in range(ROWS)]

# Check if the move is valid (i.e., if the column is not full)

def is\_valid\_move(board, col):

return board[0][col] == EMPTY

# Drop a piece into the board for the given column and player

def make\_move(board, col, player):

for row in range(ROWS - 1, -1, -1):

if board[row][col] == EMPTY:

board[row][col] = player

break

# Check if the current player has won

def is\_winner(board, player):

# Check horizontal, vertical, and diagonal

for row in range(ROWS):

for col in range(COLS):

if board[row][col] == player:

# Horizontal

if col + 3 < COLS and all(board[row][col + i] == player for i in range(4)):

return True

# Vertical

if row + 3 < ROWS and all(board[row + i][col] == player for i in range(4)):

return True

# Diagonal (\ direction)

if row + 3 < ROWS and col + 3 < COLS and all(board[row + i][col + i] == player for i in range(4)):

return True

# Diagonal (/ direction)

if row - 3 >= 0 and col + 3 < COLS and all(board[row - i][col + i] == player for i in range(4)):

return True

return False

# Evaluate the board position. Positive for maximizing player (PLAYER\_1), negative for minimizing player (PLAYER\_2)

def evaluate(board):

if is\_winner(board, PLAYER\_1):

return 1000

elif is\_winner(board, PLAYER\_2):

return -1000

return 0

# Alpha-Beta Pruning Min-Max Algorithm

def alpha\_beta(board, depth, alpha, beta, maximizing\_player):

valid\_moves = [c for c in range(COLS) if is\_valid\_move(board, c)]

# Base case: If depth is 0 or no valid moves left, return the evaluation of the board

if depth == 0 or len(valid\_moves) == 0:

return evaluate(board)

if maximizing\_player:

max\_eval = -math.inf

for move in valid\_moves:

new\_board = [row[:] for row in board] # Create a copy of the board

make\_move(new\_board, move, PLAYER\_1)

eval = alpha\_beta(new\_board, depth - 1, alpha, beta, False)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha: # Prune the branch

break

return max\_eval

else:

min\_eval = math.inf

for move in valid\_moves:

new\_board = [row[:] for row in board] # Create a copy of the board

make\_move(new\_board, move, PLAYER\_2)

eval = alpha\_beta(new\_board, depth - 1, alpha, beta, True)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha: # Prune the branch

break

return min\_eval

# Find the best move for the current player (Maximizing Player)

def find\_best\_move(board):

best\_value = -math.inf

best\_move = None

valid\_moves = [c for c in range(COLS) if is\_valid\_move(board, c)]

for move in valid\_moves:

new\_board = [row[:] for row in board] # Create a copy of the board

make\_move(new\_board, move, PLAYER\_1)

move\_value = alpha\_beta(new\_board, 5, -math.inf, math.inf, False)

if move\_value > best\_value:

best\_value = move\_value

best\_move = move

return best\_move

# Print the current game board

def print\_board(board):

for row in board:

print(" | ".join(str(cell) for cell in row))

print("\n")

# Main game loop for Connect Four

def play\_game():

board = create\_board()

game\_over = False

turn = PLAYER\_1 # Player 1 (Maximizing Player) starts

while not game\_over:

print\_board(board)

if turn == PLAYER\_1:

print("Player 1's turn (Maximizing Player)")

move = find\_best\_move(board)

print(f"Player 1 chooses column {move}")

else:

print("Player 2's turn (Minimizing Player)")

move = int(input("Enter column (0-6): "))

if is\_valid\_move(board, move):

make\_move(board, move, turn)

if is\_winner(board, turn):

print\_board(board)

print(f"Player {turn} wins!")

game\_over = True

else:

turn = PLAYER\_2 if turn == PLAYER\_1 else PLAYER\_1

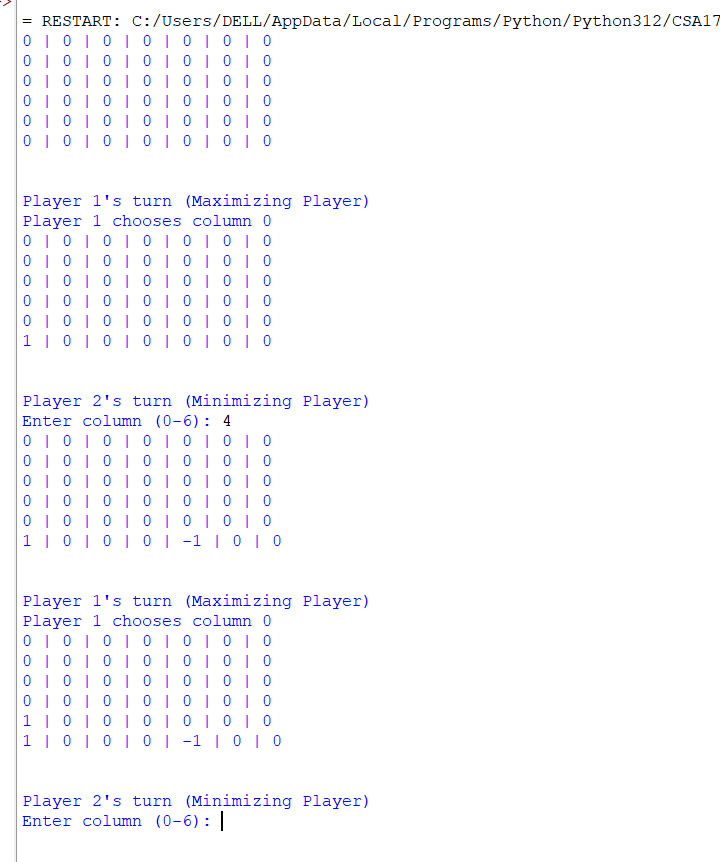
else:

print("Invalid move. Try again.")

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

play\_game()

OUTPUT



15.DECISION TREE

from sklearn import tree

clf = tree.DecisionTreeClassifier()

X = [[20, 50000, 1], [25, 70000, 0], [30, 90000, 1], [35, 120000, 0], [40, 150000, 1], [45, 180000, 0], [50, 200000, 1], [55, 220000, 0], [60, 250000, 1], [65, 280000, 0]]

y = [1, 0, 1, 0, 1, 0, 1, 0, 1, 0]

clf = clf.fit(X, y)

age = int(input("Enter your age: "))

income = int(input("Enter your income: "))

student = int(input("Are you a student? (1 for yes, 0 for no): "))

prediction = clf.predict([[age, income, student]])

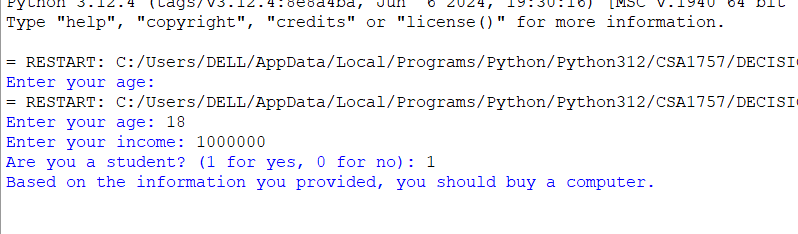
if prediction[0] == 1:

print("Based on the information you provided, you should buy a computer.")

else:

print("Based on the information you provided, you should not buy a computer.")

OUTPUT



16.ARTIFICIAL NEURAL NETWORK

import numpy as np

# Sigmoid activation function and its derivative

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

# Neural Network class

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

# Initialize weights and biases

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

# Randomly initialize weights and biases

self.weights\_input\_hidden = np.random.randn(self.input\_size, self.hidden\_size)

self.weights\_hidden\_output = np.random.randn(self.hidden\_size, self.output\_size)

self.bias\_hidden = np.zeros((1, self.hidden\_size))

self.bias\_output = np.zeros((1, self.output\_size))

# Forward propagation

def forward(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

# Backpropagation and training using gradient descent

def backward(self, X, y, learning\_rate):

# Calculate error

error = y - self.final\_output

# Calculate gradients for weights and biases

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

d\_hidden = d\_output.dot(self.weights\_hidden\_output.T) \* sigmoid\_derivative(self.hidden\_output)

# Update weights and biases using gradient descent

self.weights\_hidden\_output += self.hidden\_output.T.dot(d\_output) \* learning\_rate

self.weights\_input\_hidden += X.T.dot(d\_hidden) \* learning\_rate

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

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

# Training function

def train(self, X, y, epochs, learning\_rate):

for epoch in range(epochs):

self.forward(X) # Forward propagation

self.backward(X, y, learning\_rate) # Backpropagation

if epoch % 1000 == 0:

loss = np.mean(np.square(y - self.final\_output))

print(f"Epoch {epoch}/{epochs}, Loss: {loss:.4f}")

# Predict function

def predict(self, X):

return self.forward(X)

# Define input data (AND Gate)

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

[0, 1],

[1, 0],

[1, 1]])

# Define output data (AND Gate)

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

# Initialize Neural Network with 2 input neurons, 3 hidden neurons, and 1 output neuron

nn = NeuralNetwork(input\_size=2, hidden\_size=3, output\_size=1)

# Train the neural network for 10000 epochs with a learning rate of 0.1

nn.train(X, y, epochs=10000, learning\_rate=0.1)

# Make predictions

print("\nPredictions after training:")

predictions = nn.predict(X)

print(predictions)

OUTPUT

