Lab1:

#breadth first search

def bfs(graph, start):

    """Perform Breadth-First Search (BFS) on a graph using a list as the queue."""

    visited = set()

    queue = [start]  # Using a list instead of deque

    order = []

    while queue:

        node = queue.pop(0)  # Popping from the front of the list

        if node not in visited:

            visited.add(node)

            order.append(node)

            # Append neighbors to the queue

            for neighbor in graph.get(node, []):

                if neighbor not in visited:

                    queue.append(neighbor)

    return order

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E']

}

start\_node = 'A'

print("BFS Traversal Order:", bfs(graph, start\_node))

[Running] python -u "c:\Users\Dell\Desktop\ai\lab1breadth first search.py"

BFS Traversal Order: ['A', 'B', 'C', 'D', 'E', 'F']

[Done] exited with code=0 in 0.104 seconds

Lab2:

#2.depth first search

def dfs(graph, start, visited=None):

    """Perform Depth-First Search (DFS) on a graph using sets for neighbors."""

    if visited is None:

        visited = set()  # Initialize visited set if it's not provided

    visited.add(start)  # Mark the current node as visited

    print(start)  # Print the current node

    # Recursively visit all unvisited neighbors

    for next in graph[start] - visited:

        dfs(graph, next, visited)  # Recursively call DFS for the unvisited neighbor

    return visited

graph = {

    'A': set(['B', 'C']),

    'B': set(['A', 'D', 'E']),

    'C': set(['D', 'F','G']),

    'D': set(['B', 'E']),

    'E': set(['B', 'F', 'I']),

    'F': set(['E', 'C']),

    'G': set(['C']),  # New node 'G'

    'H': set(['D']),  # New node 'H'

    'I': set(['E'])   # New node 'I'

}

dfs(graph, 'E')

E

B

A

C

D

F

G

DFI

Lab3:

#lab3depth limit search

def dls(graph, start, depth\_limit, visited=None, depth=0):

    """Perform Depth-Limited Search (DLS) on a graph with a specified depth limit."""

    if visited is None:

        visited = set()  # Initialize visited set if it's not provided

    # If the depth limit is reached or exceeded, stop the search

    if depth > depth\_limit:

        return visited

    visited.add(start)  # Mark the current node as visited

    print(f"{' ' \* depth}{start}(Depth:{depth})")  # Indent and print the current node with its depth

    # Visit neighbors in the custom defined order

    for next\_node in graph[start] - visited:

            dls(graph, next\_node, depth\_limit, visited, depth + 1)  # Recursively call DLS for the unvisited neighbor

    return visited  # Return the visited set to propagate it back

# Define the graph with sets for adjacency lists

graph = {

    'A': set(['B', 'C']),

    'B': set(['A', 'D', 'E']),

    'C': set(['D', 'F', 'G']),

    'D': set(['B', 'E']),

    'E': set(['B', 'F', 'I']),

    'F': set(['E', 'C']),

    'G': set(['C']),  # New node 'G'

    'H': set(['D']),  # New node 'H'

    'I': set(['E'])   # New node 'I'

}

# Call Depth-Limited Search (DLS) starting from node 'E' with a depth limit of 3

dls(graph, 'E', depth\_limit=2)

E(Depth:0)

I(Depth:1)

F(Depth:1)

C(Depth:2)

B(Depth:1)

D(Depth:2)

A(Depth:2)

Lab 4:

#4.iteratitive depth limit search:

def dls(graph, start, depth\_limit, visited=None, depth=0):

    if visited is None:

        visited = set()

    # Check if the depth limit has been reached

    if depth > depth\_limit:

        return visited

    # Mark the current node as visited

    visited.add(start)

    print(f"{'  ' \* depth}{start} (Depth: {depth})")  # Print node with depth indentation

    # Recurse for each neighbor if not visited and within depth limit

    for next\_node in graph[start] - visited:

        dls(graph, next\_node, depth\_limit, visited, depth + 1)

    return visited

def iddfs(graph, start, goal, max\_depth):

    for depth in range(max\_depth + 1):

        print(f"Searching with depth limit: {depth}")

        visited = set()

        result = dls(graph, start, depth, visited)

        if goal in visited:

            print(f"Goal {goal} found at depth {depth}")

            return visited

    print(f"Goal {goal} not found within depth limit {max\_depth}")

    return visited

# Example graph

graph = {

    'A': set(['B', 'C']),

    'B': set(['A', 'D', 'E']),

    'C': set(['A', 'F', 'G']),

    'D': set(['B', 'E']),

    'E': set(['B', 'F', 'I']),

    'F': set(['E', 'C']),

    'G': set(['C']),

    'H': set(['D']),

    'I': set(['E'])

}

# Perform Iterative Deepening DFS from node 'E' with a goal 'G' and max depth of 3

iddfs(graph, 'E', 'G', max\_depth=3)

Searching with depth limit: 0

E (Depth: 0)

Searching with depth limit: 1

E (Depth: 0)

I (Depth: 1)

F (Depth: 1)

B (Depth: 1)

Searching with depth limit: 2

E (Depth: 0)

I (Depth: 1)

F (Depth: 1)

C (Depth: 2)

B (Depth: 1)

D (Depth: 2)

A (Depth: 2)

Searching with depth limit: 3

E (Depth: 0)

I (Depth: 1)

F (Depth: 1)

C (Depth: 2)

G (Depth: 3)

A (Depth: 3)

B (Depth: 1)

D (Depth: 2)

Goal G found at depth 3

Lab 5:

#lab5uiform cost search

import heapq

def uniform\_cost\_search(graph, start, goal):

    # Priority queue (min-heap), starting with the start node and a cost of 0

    priority\_queue = [(0, start)]  # Each element is a tuple (cost, node)

    # Dictionary to store the minimum cost to reach each node

    costs = {start: 0}

    # Dictionary to store the parent of each node to reconstruct the path later

    came\_from = {start: None}

    while priority\_queue:

        # Pop the node with the least cost from the priority queue

        current\_cost, current\_node = heapq.heappop(priority\_queue)

        # If the goal is reached, reconstruct the path and return it

        if current\_node == goal:

            path = []

            while current\_node:

                path.append(current\_node)

                current\_node = came\_from[current\_node]

            return path[::-1], current\_cost

        # Explore the neighbors of the current node

        if current\_cost > costs[current\_node]:

            continue

        for neighbor, cost in graph[current\_node].items():

            new\_cost = costs[current\_node] + cost

            # If this path to the neighbor is cheaper, add it to the frontier

            if neighbor not in costs or new\_cost < costs[neighbor]:

                costs[neighbor] = new\_cost

                came\_from[neighbor] = current\_node

                heapq.heappush(priority\_queue, (new\_cost, neighbor))

    # If no path is found, return None

    return None, float('inf')

# Define a sample graph where nodes have edge costs

# The graph is represented as an adjacency list with costs

graph = {

    'A': {'B': 1, 'C': 4},

    'B': {'A': 1, 'D': 2, 'E': 5},

    'C': {'A': 4, 'E': 1},

    'D': {'B': 5, 'E': 3, 'F': 1},

    'E': {'B': 5, 'C': 1, 'D': 3, 'F': 2},

    'F': {'D': 1, 'E': 2}

}

# Run the UCS from 'A' to 'F'

path, cost = uniform\_cost\_search(graph, 'A', 'F')

# Display the result

print(f"Path: {path}, Cost: {cost}")

Path: ['A', 'B', 'D', 'F'], Cost: 4

Lab6:

#lab6best first search:

import heapq

def best\_first\_search(graph, start, goal, heuristic):

    # Priority queue initialized with the start node and its heuristic

    priority\_queue = []

    heapq.heappush(priority\_queue, (heuristic[start], start))

    visited = set()

    came\_from = {start: None}

    while priority\_queue:

        current\_priority, current\_node = heapq.heappop(priority\_queue)

        if current\_node == goal:

            # Reconstruct the path from goal to start

            path = []

            while current\_node:

                path.append(current\_node)

                current\_node = came\_from[current\_node]

            return path[::-1]

        visited.add(current\_node)

        # Explore neighbors of the current node

        for neighbor in graph[current\_node]:

            if neighbor not in visited:

                heapq.heappush(priority\_queue, (heuristic[neighbor], neighbor))

            if neighbor not in came\_from:

                came\_from[neighbor] = current\_node

    return None

# Graph structure with alphabetic nodes

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F', 'G'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E'],

    'G': ['C']

}

# Heuristic values: estimated cost from each node to the goal ('G')

heuristic = {

    'A': 4,

    'B': 3,

    'C': 2,

    'D': 3,

    'E': 2,

    'F': 1,

    'G': 0  # Goal node

}

# Start and goal nodes

start = 'A'

goal = 'G'

# Run the search from start to goal

path = best\_first\_search(graph, start, goal, heuristic)

# Display the result

print(f"Path: {path}")

‘’’

graph = {

    (0, 0): [(0, 1), (1, 0)],

    (0, 1): [(0, 0), (1, 1)],

    (1, 0): [(0, 0), (1, 1)],

    (1, 1): [(0, 1), (1, 0)]

}

# Heuristic values: estimated cost from each node to the goal

heuristic = {

    (0, 0): 2,

    (0, 1): 1,

    (1, 0): 1,

    (1, 1): 0  # Goal node

}

Path: ['A', 'C', 'G']’’’

Lab7:

#lab7beam search algorithm

import heapq

def beam\_search(graph, start, goal, heuristic, beam\_width):

    # Priority queue initialized with the start node and its heuristic value

    frontier = [(heuristic[start], start)]

    visited = set([start])  # Set to track visited nodes

    came\_from = {start: None}  # Dictionary to track the predecessors of nodes

    while frontier:

        # Sort the frontier based on the heuristic value

        frontier = sorted(frontier, key=lambda x: x[0])

        # Limit the number of nodes in the frontier based on the beam width

        frontier = frontier[:beam\_width]

        next\_frontier = []

        # Explore each node in the frontier

        for \_, current\_node in frontier:

            if current\_node == goal:

                # Reconstruct the path if the goal is reached

                path = []

                while current\_node is not None:

                    path.append(current\_node)

                    current\_node = came\_from[current\_node]

                return path[::-1]

            # Add the neighbors of the current node to the next frontier

            for neighbor in graph[current\_node]:

                if neighbor not in visited:

                    visited.add(neighbor)

                    came\_from[neighbor] = current\_node  # Set the current node as the predecessor of the neighbor

                    next\_frontier.append((heuristic[neighbor], neighbor))

        # Update the frontier for the next iteration

        frontier = next\_frontier

    return None  # Return None if no path is found

# Graph structure with alphabetic nodes

graph = {

    'A': ['B', 'C'],

    'B': ['A', 'D', 'E'],

    'C': ['A', 'F', 'G'],

    'D': ['B'],

    'E': ['B', 'F'],

    'F': ['C', 'E'],

    'G': ['C']

}

# Heuristic values: estimated cost from each node to the goal ('G')

heuristic = {

    'A': 4,

    'B': 3,

    'C': 2,

    'D': 3,

    'E': 2,

    'F': 1,

    'G': 0  # Goal node

}

# Start and goal nodes

start = 'A'

goal = 'G'

# Define the beam width

beam\_width = 2

# Run the Beam Search from start to goal

path = beam\_search(graph, start, goal, heuristic, beam\_width)

# Display the result

print(f"Path: {path}")

Path: ['A', 'C', 'G']

Lab8:

#lab8hill climbe algorithm

def hill\_climbing(graph, start, goal, heuristic):

    # Initialize the current node and its cost (heuristic value)

    current\_node = start

    current\_cost = heuristic[current\_node]

    path = [current\_node]

    while current\_node != goal:

        # Get the neighbors of the current node

        neighbors = graph.get(current\_node, [])

        # If there are no neighbors, stop the search

        if not neighbors:

            break

        # Sort the neighbors based on their heuristic value in descending order

        neighbors.sort(key=lambda x: heuristic[x[0]])

        # Get the best neighbor with the highest heuristic value

        next\_node, cost = neighbors[0]

        next\_cost = heuristic[next\_node]

        # If the next node does not improve the heuristic, stop (stuck at a local maximum)

        if next\_cost >= current\_cost:

            break

        # Move to the next node

        current\_node = next\_node

        current\_cost = next\_cost

        path.append(current\_node)

    # If the goal is reached, return the path, otherwise return None

    return path if current\_node == goal else None

# Define the graph with nodes, neighbors, and edge costs

graph = {

    'A': [('B', 1), ('C', 4)],

    'B': [('D', 2), ('E', 5)],

    'C': [('F', 3)],

    'D': [('G', 6)],

    'E': [('G', 2)],

    'F': [('G', 1)],

    'G': []

}

# Define the heuristic values for each node

heuristic = {

    'A': 7,

    'B': 6,

    'C': 4,

    'D': 3,

    'E': 2,

    'F': 1,

    'G': 0

}

# Start node and goal node

start = 'A'

goal = 'G'

# Run the hill climbing algorithm

path = hill\_climbing(graph, start, goal, heuristic)

# Display the result

if path:

    print(f"Path found: {' -> '.join(path)}")

else:

    print("No path found or stuck in local optimum")

Path found: A -> C -> F -> G

Lab9:

#lab9 A\* algorithm

import heapq

def a\_star(graph, start, goal, heuristic):

    # Priority queue to store nodes with their f(n) values (f(n) = g(n) + h(n))

    open\_list = []

    heapq.heappush(open\_list, (0 + heuristic[start], start))

    # Dictionaries to store the actual cost to reach a node and the parent of each node

    g\_costs = {start: 0}  # g(n) is the cost from the start node to the current node

    came\_from = {start: None}  # To reconstruct the path

    while open\_list:

        # Pop the node with the lowest f(n) from the priority queue

        \_, current\_node = heapq.heappop(open\_list)

        # If the goal is reached, reconstruct the path and return it

        if current\_node == goal:

            path = []

            while current\_node:

                path.append(current\_node)

                current\_node = came\_from[current\_node]

            return path[::-1]  # Reverse the path to get the correct order

        # Explore the neighbors of the current node

        for neighbor, cost in graph[current\_node]:

            # Calculate g(n) for the neighbor

            tentative\_g\_cost = g\_costs[current\_node] + cost

            # If this neighbor has not been visited or a shorter path is found

            if neighbor not in g\_costs or tentative\_g\_cost < g\_costs[neighbor]:

                g\_costs[neighbor] = tentative\_g\_cost

                f\_cost = tentative\_g\_cost + heuristic[neighbor]

                came\_from[neighbor] = current\_node

                heapq.heappush(open\_list, (f\_cost, neighbor))  # Add the neighbor to the open list

    # If the goal is not reached, return None

    return None

# Define the graph with nodes, neighbors, and edge costs

graph = {

    'A': [('B', 1), ('C', 4)],

    'B': [('D', 2), ('E', 5)],

    'C': [('F', 3)],

    'D': [('G', 6)],

    'E': [('G', 2)],

    'F': [('G', 1)],

    'G': []

}

# Define the heuristic values for each node

heuristic = {

    'A': 7,  # Heuristic value for node A

    'B': 6,  # Heuristic value for node B

    'C': 4,  # Heuristic value for node C

    'D': 3,  # Heuristic value for node D

    'E': 2,  # Heuristic value for node E

    'F': 1,  # Heuristic value for node F

    'G': 0   # Heuristic value for node G (goal node)

}

# Start node and goal node

start = 'A'

goal = 'G'

# Run the A\* algorithm

path = a\_star(graph, start, goal, heuristic)

# Display the result

if path:

    print(f"Path found: {' -> '.join(path)}")

else:

    print("No path found")

Path found: A -> B -> E -> G

Lab10:

#lab10

# Rule-based system

def rule\_based\_system(facts):

    rules = [

        ("if it is raining, then take an umbrella", lambda facts: "raining" in facts),

        ("if it is cold, then wear a coat", lambda facts: "cold" in facts),

        ("if you are going outside and it is raining, then take an umbrella", lambda facts: "going\_outside" in facts and "raining" in facts),

    ]

    actions = []

    for rule, condition in rules:

        if condition(facts):

            actions.append(rule)

    return actions

# Semantic network and Animal class

class Animal:

    def \_\_init\_\_(self, name, species, sound):

        self.name = name

        self.species = species

        self.sound = sound

        self.network = {}

    def add\_relationship(self, parent, child):

        if parent not in self.network:

            self.network[parent] = []

        self.network[parent].append(child)

    def display(self):

        for parent, children in self.network.items():

            for child in children:

                print(f"{parent} → {child}")

    def make\_sound(self):

        return self.sound

class SemanticNetwork:

    def \_\_init\_\_(self):

        self.network = {}

    def add\_relationship(self, parent, child):

        if parent not in self.network:

            self.network[parent] = []

        self.network[parent].append(child)

    def display(self):

        for parent, children in self.network.items():

            for child in children:

                print(f"{parent} → {child}")

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

    # Rule-based system test

    facts = ["raining", "going\_outside"]

    actions = rule\_based\_system(facts)

    print("Rule-based Actions:", actions)

    # Adding a predicate logic action print

    if "if it is raining, then take an umbrella" in actions:

        print("Predicate Logic Action: Take an umbrella")

    elif "if it is cold, then wear a coat" in actions:

        print("Predicate Logic Action: Wear a coat")

    elif "if you are going outside and it is raining, then take an umbrella" in actions:

        print("Predicate Logic Action: Take an umbrella")

    else:

        print("Predicate Logic Action: No action needed")

    # Semantic network implementation

    print("\nSemantic Network Relationships:")

    dog = Animal("Buddy", "dog", "woof")

    cat = Animal("Whiskers", "cat", "meow")

    print(f"{dog.name} says: {dog.make\_sound()}")

    print(f"{cat.name} says: {cat.make\_sound()}")

    # Semantic network relationships

    semantic\_net = SemanticNetwork()

    semantic\_net.add\_relationship("Animal", "Dog")

    semantic\_net.add\_relationship("Animal", "Cat")

    semantic\_net.add\_relationship("Dog", "Barks")

    semantic\_net.add\_relationship("Cat", "Meows")

    print("\nSemantic Network:")

    semantic\_net.display()

Rule-based Actions: ['if it is raining, then take an umbrella', 'if you are going outside and it is raining, then take an umbrella']

Predicate Logic Action: Take an umbrella

Semantic Network Relationships:

Buddy says: woof

Whiskers says: meow

Semantic Network:

Animal → Dog

Animal → Cat

Dog → Barks

Cat → Meows

Lab 11:

# Importing necessary libraries

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn import datasets

import matplotlib.pyplot as plt

# Loading the Iris dataset

iris = datasets.load\_iris()

# Plotting the first two features of the Iris dataset

\_, ax = plt.subplots()

scatter = ax.scatter(iris.data[:, 0], iris.data[:, 1], c=iris.target)

ax.set(xlabel=iris.feature\_names[0], ylabel=iris.feature\_names[1])

# Adding a legend to the scatter plot

\_ = ax.legend(

    scatter.legend\_elements()[0], iris.target\_names, loc="lower right", title="Classes"

)

# Features and labels

X = iris.data  # Features

y = iris.target  # Labels

# Splitting the dataset into training and testing sets (70% train, 30% test)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Creating a Naive Bayes classifier (GaussianNB for continuous data)

nb\_classifier = GaussianNB()

# Training the model on the training data

nb\_classifier.fit(X\_train, y\_train)

# Making predictions on the test data

y\_pred = nb\_classifier.predict(X\_test)

# Evaluating the model

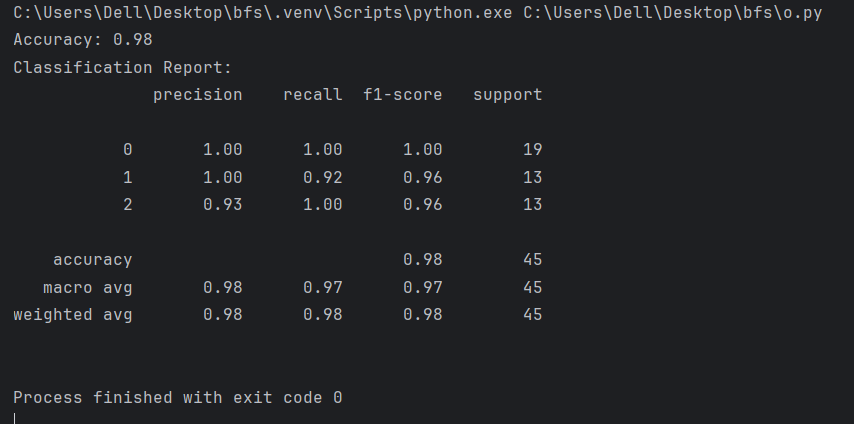
accuracy = accuracy\_score(y\_test, y\_pred)

# Displaying results

print(f'Accuracy: {accuracy:.2f}')

print('Classification Report:')

print(classification\_report(y\_test, y\_pred))



Lab12:

import numpy as np

# Sigmoid activation function

def sigmoid(x):

    return 1 / (1 + np.exp(-x))

# Derivative of sigmoid function

def sigmoid\_derivative(x):

    return x \* (1 - x)

# Neural network class

class NeuralNetwork:

    def \_\_init\_\_(self):

        # Initialize weights and bias randomly

        self.weights = np.random.rand(2, 1)  # Two inputs (AND/OR gate)

        self.bias = np.random.rand(1)

        self.learning\_rate = 0.1

    # Train the network with inputs and expected output

    def train(self, inputs, expected\_output, epochs):

        for epoch in range(epochs):

            total\_error = 0

            for x, y in zip(inputs, expected\_output):

                # Feed forward

                weighted\_sum = np.dot(x, self.weights) + self.bias

                activated\_output = sigmoid(weighted\_sum)

                # Calculate the error

                error = y - activated\_output

                total\_error += error \*\* 2  # Sum of squared errors for monitoring

                # Backpropagation

                adjustments = error \* sigmoid\_derivative(activated\_output)

                # Update weights and bias

                self.weights += self.learning\_rate \* np.dot(x.reshape(-1, 1), adjustments.reshape(1, -1))

                self.bias += self.learning\_rate \* adjustments

            if epoch % 1000 == 0:  # Print error every 1000 epochs

                print(f"Epoch {epoch}/{epochs}, Total Error: {total\_error}")

    # Predict the output for a given input

    def predict(self, input\_data):

        weighted\_sum = np.dot(input\_data, self.weights) + self.bias

        activated\_output = sigmoid(weighted\_sum)

        return np.round(activated\_output)  # Return 0 or 1 (binary output)

# Define input and output for AND gate

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

and\_outputs = np.array([[0], [0], [0], [1]])

# Define input and output for OR gate

or\_outputs = np.array([[0], [1], [1], [1]])

# Initialize the neural network

and\_gate\_nn = NeuralNetwork()

or\_gate\_nn = NeuralNetwork()

# Train the network for AND gate

print("Training Neural Network for AND gate...")

and\_gate\_nn.train(inputs, and\_outputs, 10000)

# Train the network for OR gate

print("Training Neural Network for OR gate...")

or\_gate\_nn.train(inputs, or\_outputs, 10000)

# Testing AND gate

print("\nAND Gate Results:")

for input\_data in inputs:

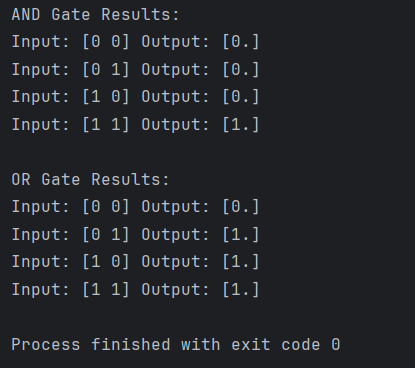
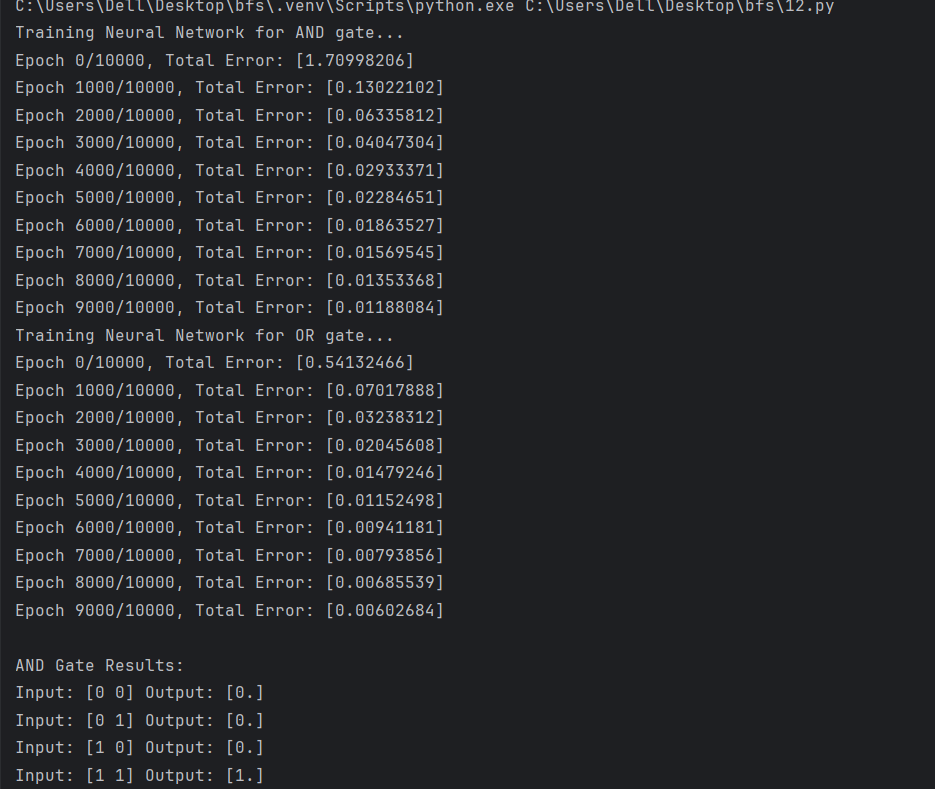
    print(f"Input: {input\_data} Output: {and\_gate\_nn.predict(input\_data)}")

# Testing OR gate

print("\nOR Gate Results:")

for input\_data in inputs:

    print(f"Input: {input\_data} Output: {or\_gate\_nn.predict(input\_data)}")



Lab 13:

#lab13: Back propagation

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)

# NeuralNetwork class

class NeuralNetwork:

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

        # Initialize weights and biases

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

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

        self.learning\_rate = 0.5

    def forward(self, x):

        # Forward propagation

        self.input = x

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

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

        return self.output

    def backward(self, x, y, output):

        # Backpropagation

        output\_error = y - output

        output\_delta = output\_error \* sigmoid\_derivative(output)

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

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

        # Update weights

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

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

    def train(self, x, y, iterations):

        for i in range(iterations):

            output = self.forward(x)

            self.backward(x, y, output)

            if i % 1000 == 0:

                loss = np.mean(np.square(y - output))

                print(f"Iteration {i}, Loss: {loss}")

# Example usage

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

    # XOR dataset

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

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

    # Initialize and train the network

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

    nn.train(x, y, iterations=10000)

    # Test the network

    print("\nAfter training:")

    print(nn.forward(x))

teration 0, Loss: 0.2503962271218796

Iteration 1000, Loss: 0.2499226390431713

Iteration 2000, Loss: 0.1722431066483968

Iteration 3000, Loss: 0.12261154906996766

Iteration 4000, Loss: 0.1107783109200159

Iteration 5000, Loss: 0.1048855494652917

Iteration 6000, Loss: 0.10116233156130244

Iteration 7000, Loss: 0.09851928755868186

Iteration 8000, Loss: 0.09650813230333344

Iteration 9000, Loss: 0.09490552303571176

After training:

[[0.05924071]

[0.83246595]

[0.83249847]

[0.56098946]]

Lab14:

#lab14weather expert system:

class WeatherExpertSystem:

    def init(self, temperature, humidity, is\_cloudy, is\_sunny):

        self.temperature = temperature

        self.humidity = humidity

        self.is\_cloudy = is\_cloudy

        self.is\_sunny = is\_sunny

    def predict\_weather(self):

        if self.temperature > 30 and self.humidity < 40:

            return "Prediction: Hot and dry."

        elif self.temperature < 10:

            return "Prediction: Cold weather."

        elif self.is\_cloudy and self.humidity > 60:

            return "Prediction: Rain likely."

        elif 20 <= self.temperature <= 30 and self.humidity < 50:

            return "Prediction: Pleasant weather."

        elif self.is\_sunny:

            return "Prediction: Sunny weather."

        else:

            return "Prediction: Uncertain conditions."

# Input section

temperature = float(input("Enter the temperature: "))

humidity = float(input("Enter the humidity: "))

is\_cloudy = input("Is it cloudy? (yes/no): ").lower() == "yes"

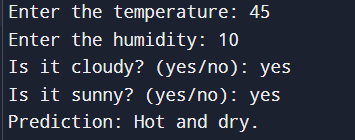
is\_sunny = input("Is it sunny? (yes/no): ").lower() == "yes"

# Prediction

weather = WeatherExpertSystem(temperature, humidity, is\_cloudy,is\_sunny)

prediction = weather.predict\_weather()

print(prediction)



Lab15:

#lab15:N queen problem

def is\_safe(board, row, col, N):

    # Check left side in the current row

    for i in range(col):

        if board[row][i] == 1:

            return False

    # Check upper-left diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    # Check lower-left diagonal

    for i, j in zip(range(row, N), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    return True

def solve\_n\_queens\_util(board, col, N):

    # If all queens are placed, return True

    if col >= N:

        return True

    # Try placing the queen in all rows of the current column

    for i in range(N):

        if is\_safe(board, i, col, N):

            board[i][col] = 1  # Place the queen

            # Recurse for the next column

            if solve\_n\_queens\_util(board, col + 1, N):

                return True

            board[i][col] = 0  # Backtrack

    return False

def solve\_n\_queens(N):

    # Initialize the board

    board = [[0 for \_ in range(N)] for \_ in range(N)]

    if not solve\_n\_queens\_util(board, 0, N):

        print("Solution does not exist.")

        return

    # Print the solution

    for row in board:

        print(" ".join("Q" if x == 1 else "." for x in row))

    print()

# Input from the user

N = int(input("Enter the value of N for the N-Queen problem: "))

solve\_n\_queens(N)

def is\_safe(board, row, col, N):

    # Check left side in the current row

    for i in range(col):

        if board[row][i] == 1:

            return False

    # Check upper-left diagonal

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    # Check lower-left diagonal

    for i, j in zip(range(row, N), range(col, -1, -1)):

        if board[i][j] == 1:

            return False

    return True

def solve\_n\_queens\_util(board, col, N):

    # If all queens are placed, return True

    if col >= N:

        return True

    # Try placing the queen in all rows of the current column

    for i in range(N):

        if is\_safe(board, i, col, N):

            board[i][col] = 1  # Place the queen

            # Recurse for the next column

            if solve\_n\_queens\_util(board, col + 1, N):

                return True

            board[i][col] = 0  # Backtrack

    return False

def solve\_n\_queens(N):

    # Initialize the board

    board = [[0 for \_ in range(N)] for \_ in range(N)]

    if not solve\_n\_queens\_util(board, 0, N):

        print("Solution does not exist.")

        return

    # Print the solution

    for row in board:

        print(" ".join("Q" if x == 1 else "." for x in row))

    print()

# Input from the user

N = int(input("Enter the value of N for the N-Queen problem: "))

solve\_n\_queens(N)

Enter the value of N for the N-Queen problem: 8

Q . . . . . . .

. . . . . . Q .

. . . . Q . . .

. . . . . . . Q

. Q . . . . . .

. . . Q . . . .

. . . . . Q . .

. . Q . . . . .

Lab16:

#lab 16:minmax

import math

def minmax(depth, nodeindex, isMax, scores, h):

    if depth == h:

        return scores[nodeindex]

    if isMax:

        return max(

            minmax(depth + 1, nodeindex \* 2, False, scores, h),

            minmax(depth + 1, nodeindex \* 2 + 1, False, scores, h)

        )

    else:

        return min(

            minmax(depth + 1, nodeindex \* 2, True, scores, h),

            minmax(depth + 1, nodeindex \* 2 + 1, True, scores, h)

        )

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

    scores = [3, 5, 2, 9, 12, 5, 23, 2, 3]

    h = int(math.log2(len(scores)))  # Using log2 for binary trees

    result = minmax(0, 0, True, scores, h)

    print(f"The optimal value is: {result}")

The optimal value is: 12