|  |
| --- |
| **1)Implement Recursive Depth First Search Algorithm. Read the undirected unweighted graph from a .csv file.**  import pandas as pd  # Step 1: Read graph from CSV  data = pd.read\_csv(r'D:\------------------------------\SEM VI\AI LAB\Practical Exam\exp1.csv') # Ensure correct file path  graph = {}  # Create adjacency list (graph) from CSV data  for index, row in data.iterrows():  if row['Source'] not in graph:  graph[row['Source']] = []  if row['Destination'] not in graph:  graph[row['Destination']] = []  graph[row['Source']].append(row['Destination'])  graph[row['Destination']].append(row['Source'])  # Step 2: Define the DFS function (Recursive)  visited = set() # Set to track visited nodes  def dfs\_recursive(graph, node):  if node not in visited:  print(node, end=' ') # Print the node  visited.add(node) # Mark the node as visited  for neighbor in graph[node]: # Visit each neighbor  dfs\_recursive(graph, neighbor)  # Step 3: Ask the user for the starting node  start\_node = input("Enter the start node for DFS traversal: ")  # Ensure the start node exists in the graph  if start\_node in graph:  print("DFS Traversal Order: ", end='')  dfs\_recursive(graph, start\_node)  else:  print(f"Node {start\_node} does not exist in the graph.")  **Enter the start node for DFS traversal: A**  **DFS Traversal Order: A B D C E F** |
| **2) Implement Non-Recursive Depth First Search Algorithm. Read the undirected unweighted graph from user.**  # Function to implement Non-Recursive DFS  def non\_recursive\_dfs(graph, start\_node):  visited = set()  stack = [start\_node]  dfs\_order = []  while stack:  node = stack.pop()  if node not in visited:  visited.add(node)  dfs\_order.append(node)  # Push all unvisited neighbors to the stack  # We reverse the neighbors to visit them in the correct order (left to right)  stack.extend(reversed(graph[node]))  return dfs\_order  # Input: Read graph from the user  def read\_graph():  graph = {}  n = int(input("Enter the number of nodes: "))  # Read the edges and construct the adjacency list  for \_ in range(n):  node = input("Enter node name: ")  graph[node] = []  m = int(input("Enter the number of edges: "))  for \_ in range(m):  u, v = input("Enter edge (u, v): ").split()  if v not in graph[u]:  graph[u].append(v)  if u not in graph[v]:  graph[v].append(u)  return graph  # Main program  graph = read\_graph()  start\_node = input("Enter the start node for DFS traversal: ")  # Perform Non-Recursive DFS  dfs\_order = non\_recursive\_dfs(graph, start\_node)  print(f"DFS Traversal Order: {' '.join(dfs\_order)}")  **Enter the number of nodes: 5**  **Enter node name: A**  **Enter node name: B**  **Enter node name: C**  **Enter node name: D**  **Enter node name: E**  **Enter the number of edges: 5**  **Enter edge (u, v): A B**  **Enter edge (u, v): A C**  **Enter edge (u, v): B D**  **Enter edge (u, v): C E**  **Enter edge (u, v): D E**  **Enter the start node for DFS traversal: A**  **DFS Traversal Order: A B D E C** |
| **3) Implement Breadth First Search Algorithm. Read the undirected unweighted graph from user.**  from collections import deque  def bfs(graph, start):  visited = set() # Set to track visited nodes  queue = deque([start]) # Queue to store nodes to visit  bfs\_order = [] # To store the order of traversal  while queue:  node = queue.popleft() # Pop from front of the queue  if node not in visited:  visited.add(node) # Mark the node as visited  bfs\_order.append(node) # Add node to BFS order  # Enqueue all the unvisited neighbors of the node  for neighbor in graph[node]:  if neighbor not in visited:  queue.append(neighbor)  return bfs\_order  # Read input from user for graph  def create\_graph():  graph = {}  print("Enter edges in the format: node1 node2")  print("Type 'done' when you are finished.")  while True:  edge = input("Enter edge: ").strip()  if edge.lower() == 'done':  break    node1, node2 = edge.split()    if node1 not in graph:  graph[node1] = []  if node2 not in graph:  graph[node2] = []  # Add undirected edges  graph[node1].append(node2)  graph[node2].append(node1)  return graph  # Main driver code  if \_\_name\_\_ == "\_\_main\_\_":  # Create graph based on user input  graph = create\_graph()  # Input the start node for BFS traversal  start\_node = input("Enter the start node for BFS traversal: ").strip()  # Perform BFS  bfs\_result = bfs(graph, start\_node)    # Output the BFS traversal order  print("BFS Traversal Order:", ' '.join(bfs\_result)) |
| **4) mplement Best First Search Algorithm. Read the directed unweighted graph and the heuristic values from user.**  import heapq  # Function to perform Best First Search  def best\_first\_search(graph, heuristics, start, goal):  # Priority Queue for nodes to explore  open\_list = []    # Initialize the open list with the start node and its heuristic value  heapq.heappush(open\_list, (heuristics[start], start)) # (heuristic\_value, node)    # Set to keep track of visited nodes  visited = set()    while open\_list:  # Pop the node with the lowest heuristic value  \_, current\_node = heapq.heappop(open\_list)    # If we reach the goal node, return the path  if current\_node == goal:  print(f"Goal node {goal} found!")  return    # Mark the node as visited  visited.add(current\_node)    print(f"Visiting node: {current\_node}")    # Explore neighbors of the current node  for neighbor in graph[current\_node]:  if neighbor not in visited:  # Push neighbors with their heuristic values into the queue  heapq.heappush(open\_list, (heuristics[neighbor], neighbor))    # Read the directed graph and heuristic values from the user  def read\_input():  graph = {}  heuristics = {}    # Read number of nodes and edges  n = int(input("Enter the number of nodes: "))    # Read graph edges  print("Enter the edges in the format 'start end' (enter 'done' when finished):")  for \_ in range(n):  node = input("Enter node name: ")  graph[node] = []    while True:  edge = input("Enter edge (or type 'done' to finish): ")  if edge.lower() == 'done':  break  start, end = edge.split()  graph[start].append(end)  # Read heuristic values for each node  print("Enter the heuristic values for each node:")  for node in graph:  heuristic\_value = int(input(f"Heuristic value for node {node}: "))  heuristics[node] = heuristic\_value    # Read start and goal nodes  start = input("Enter the start node: ")  goal = input("Enter the goal node: ")    return graph, heuristics, start, goal  # Main Function  def main():  # Read graph and heuristic values  graph, heuristics, start, goal = read\_input()    # Call Best First Search  best\_first\_search(graph, heuristics, start, goal)  if \_\_name\_\_ == "\_\_main\_\_":  main()  **Enter the number of nodes: 5**  **Enter the edges in the format 'start end' (enter 'done' when finished):**  **Enter node name: A**  **Enter node name: B**  **Enter node name: C**  **Enter node name: D**  **Enter node name: E**  **Enter edge (or type 'done' to finish): A B**  **Enter edge (or type 'done' to finish): A C**  **Enter edge (or type 'done' to finish): B D**  **Enter edge (or type 'done' to finish): C D**  **Enter edge (or type 'done' to finish): D E**  **Enter edge (or type 'done' to finish): done**  **Enter the heuristic values for each node:**  **Heuristic value for node A: 5**  **Heuristic value for node B: 2**  **Heuristic value for node C: 4**  **Heuristic value for node D: 1**  **Heuristic value for node E: 3**  **Enter the start node: A**  **Enter the goal node: E**  **Visiting node: A**  **Visiting node: B**  **Visiting node: D**  **Goal node E found!** |
| **5) Implement Best First Search Algorithm. Read the undirected weighted graph and the heuristic values from user.**  import heapq  def best\_first\_search(graph, heuristics, start, goal):  open\_list = []  heapq.heappush(open\_list, (heuristics[start], start))  visited = set()  while open\_list:  \_, current = heapq.heappop(open\_list)  if current == goal:  print(f"Goal node {goal} found!")  return  if current in visited:  continue  print(f"Visiting node: {current}")  visited.add(current)  for neighbor, weight in graph[current]:  if neighbor not in visited:  heapq.heappush(open\_list, (heuristics[neighbor], neighbor))  def read\_input():  graph = {}  heuristics = {}  n = int(input("Enter number of nodes: "))  print("Enter node names:")  nodes = []  for \_ in range(n):  node = input()  nodes.append(node)  graph[node] = []  print("Enter edges in format 'start end weight' (type 'done' to finish):")  while True:  edge = input()  if edge.lower() == 'done':  break  u, v, w = edge.split()  w = int(w)  graph[u].append((v, w))  graph[v].append((u, w)) # undirected graph  print("Enter heuristic values for each node:")  for node in nodes:  heuristics[node] = int(input(f"Heuristic for {node}: "))  start = input("Enter start node: ")  goal = input("Enter goal node: ")  return graph, heuristics, start, goal  def main():  graph, heuristics, start, goal = read\_input()  best\_first\_search(graph, heuristics, start, goal)  if \_\_name\_\_ == "\_\_main\_\_":  main()  **Enter number of nodes: 5**  **Enter node names:**  **A**  **B**  **C**  **D**  **E**  **Enter edges in format 'start end weight' (type 'done' to finish):**  **A B 1**  **A C 3**  **B D 1**  **C E 6**  **D E 2**  **done**  **Enter heuristic values for each node:**  **Heuristic for A: 5**  **Heuristic for B: 3**  **Heuristic for C: 4**  **Heuristic for D: 1**  **Heuristic for E: 0**  **Enter start node: A**  **Enter goal node: E**  **Visiting node: A**  **Visiting node: B**  **Visiting node: D**  **Goal node E found!** |
| **6) Implement Best First Search Algorithm. Read the undirected unweighted graph and the heuristic values from user.**  import heapq  # Step 1: Read the graph  graph = {}  n = int(input("Enter the number of edges: "))  for \_ in range(n):  u, v = input("Enter edge (format: u v): ").split()  if u not in graph:  graph[u] = []  if v not in graph:  graph[v] = []  graph[u].append(v)  graph[v].append(u) # undirected graph  # Step 2: Read heuristic values  heuristic = {}  nodes = graph.keys()  print("Enter heuristic values for each node:")  for node in nodes:  heuristic[node] = int(input(f"Heuristic for {node}: "))  # Step 3: Best First Search  def best\_first\_search(start, goal):  visited = set()  pq = []  heapq.heappush(pq, (heuristic[start], start))  while pq:  \_, current = heapq.heappop(pq)  print(f"Visiting node: {current}")  if current == goal:  print("Goal node", goal, "found!")  return  visited.add(current)  for neighbor in graph[current]:  if neighbor not in visited:  heapq.heappush(pq, (heuristic[neighbor], neighbor))  print("Goal node not reachable.")  start\_node = input("Enter start node: ")  goal\_node = input("Enter goal node: ")  best\_first\_search(start\_node, goal\_node)  **Enter the number of edges: 5**  **Enter edge (format: u v): A B**  **Enter edge (format: u v): A C**  **Enter edge (format: u v): B D**  **Enter edge (format: u v): C E**  **Enter edge (format: u v): D E**  **Enter heuristic values for each node:**  **Heuristic for A: 5**  **Heuristic for B: 3**  **Heuristic for C: 4**  **Heuristic for D: 1**  **Heuristic for E: 0**  **Enter start node: A**  **Enter goal node: E**  **Visiting node: A**  **Visiting node: B**  **Visiting node: D**  **Visiting node: E**  **Goal node E found!** |
| **7)** **Implement Best First Search Algorithm. Read the directed weighted graph and the heuristic values from user.**  import heapq  # Step 1: Read the directed weighted graph  graph = {}  n = int(input("Enter the number of edges: "))  for \_ in range(n):  u, v, w = input("Enter edge (format: u v weight): ").split()  w = int(w)  if u not in graph:  graph[u] = []  graph[u].append((v, w))  # Step 2: Read heuristic values  heuristic = {}  nodes = set(graph.keys())  for u in graph:  for v, \_ in graph[u]:  nodes.add(v)  print("Enter heuristic values for each node:")  for node in nodes:  heuristic[node] = int(input(f"Heuristic for {node}: "))  # Step 3: Best First Search  def best\_first\_search(start, goal):  visited = set()  pq = []  heapq.heappush(pq, (heuristic[start], start))  while pq:  \_, current = heapq.heappop(pq)  print(f"Visiting node: {current}")  if current == goal:  print(f"Goal node {goal} found!")  return  visited.add(current)  for neighbor, \_ in graph.get(current, []):  if neighbor not in visited:  heapq.heappush(pq, (heuristic[neighbor], neighbor))  print("Goal node not reachable.")  start\_node = input("Enter start node: ")  goal\_node = input("Enter goal node: ")  best\_first\_search(start\_node, goal\_node)  **Enter the number of edges: 5**  **Enter edge (format: u v weight): A B 2**  **Enter edge (format: u v weight): A C 3**  **Enter edge (format: u v weight): B D 1**  **Enter edge (format: u v weight): C E 4**  **Enter edge (format: u v weight): D E 2**  **Enter heuristic values for each node:**  **Heuristic for C: 5**  **Heuristic for E: 3**  **Heuristic for A: 4**  **Heuristic for B: 2**  **Heuristic for D: 0**  **Enter start node: A**  **Enter goal node: E**  **Visiting node: A**  **Visiting node: B**  **Visiting node: D**  **Visiting node: E**  **Goal node E found!** |
| **8)** **Implement A\* algorithm. Read directed weighted graph and heuristic values from a .csv file.**  import pandas as pd  import heapq  # Step 1: Read Graph and Heuristics from CSV  graph\_data = pd.read\_csv("D:\------------------------------\SEM VI\AI LAB\Practical Exam\exp\_8.csv") # Your CSV file  graph = {}  for index, row in graph\_data.iterrows():  src = row['Source']  dest = row['Destination']  cost = row['Cost']  heuristic = row['Heuristic']  if src not in graph:  graph[src] = []  graph[src].append((dest, cost))  # Read heuristics separately  heuristics = {}  for index, row in graph\_data.iterrows():  heuristics[row['Source']] = row['Heuristic']  heuristics[row['Destination']] = row['Heuristic'] # Make sure destination heuristic is added too  # Step 2: A\* Algorithm  def astar(graph, heuristics, start, goal):  open\_list = []  heapq.heappush(open\_list, (heuristics[start], start))  came\_from = {}  g\_cost = {start: 0}  while open\_list:  current\_f, current\_node = heapq.heappop(open\_list)  if current\_node == goal:  path = []  while current\_node in came\_from:  path.append(current\_node)  current\_node = came\_from[current\_node]  path.append(start)  path.reverse()  return path  for neighbor, cost in graph.get(current\_node, []):  tentative\_g = g\_cost[current\_node] + cost  if neighbor not in g\_cost or tentative\_g < g\_cost[neighbor]:  came\_from[neighbor] = current\_node  g\_cost[neighbor] = tentative\_g  f\_cost = tentative\_g + heuristics.get(neighbor, 0)  heapq.heappush(open\_list, (f\_cost, neighbor))  return None  # Step 3: Get Start and Goal from user  start\_node = input("Enter start node: ")  goal\_node = input("Enter goal node: ")  path = astar(graph, heuristics, start\_node, goal\_node)  if path:  print("A\* Path:", ' -> '.join(path))  else:  print("No path found.")  **Enter start node: A**  **Enter goal node: F**  **A\* Path: A -> B -> D -> E -> F** |

|  |
| --- |
| **9)Implement A\* algorithm. Read directed weighted graph and heuristic values from user.**  import heapq  def a\_star(graph, heuristics, start, goal):  open\_list = []  heapq.heappush(open\_list, (heuristics[start], start, [start], 0)) # (f = g + h, node, path, g)  visited = set()  while open\_list:  f, current, path, g = heapq.heappop(open\_list)  if current == goal:  print(f"A\* Path: {' -> '.join(path)}")  return  visited.add(current)  for neighbor, cost in graph.get(current, []):  if neighbor not in visited:  total\_cost = g + cost  heapq.heappush(open\_list, (total\_cost + heuristics[neighbor], neighbor, path + [neighbor], total\_cost))  print("Goal not reachable.")  # Input graph from user  graph = {}  num\_edges = int(input("Enter number of edges: "))  for \_ in range(num\_edges):  src = input("Enter source node: ")  dest = input("Enter destination node: ")  cost = int(input(f"Enter cost from {src} to {dest}: "))  if src not in graph:  graph[src] = []  graph[src].append((dest, cost))  # Input heuristics  heuristics = {}  nodes = set(graph.keys())  for edges in graph.values():  for dest, \_ in edges:  nodes.add(dest)  print("\nEnter heuristic values for each node:")  for node in nodes:  heuristics[node] = int(input(f"Heuristic for {node}: "))  # Input start and goal  start = input("\nEnter start node: ")  goal = input("Enter goal node: ")  # Run A\* Algorithm  a\_star(graph, heuristics, start, goal)  **Enter number of edges: 6**  **Enter source node: A**  **Enter destination node: B**  **Enter cost from A to B: 1**  **Enter source node: A**  **Enter destination node: C**  **Enter cost from A to C: 4**  **Enter source node: B**  **Enter destination node: D**  **Enter cost from B to D: 2**  **Enter source node: C**  **Enter destination node: D**  **Enter cost from C to D: 1**  **Enter source node: D**  **Enter destination node: E**  **Enter cost from D to E: 5**  **Enter source node: E**  **Enter destination node: F**  **Enter cost from E to F: 2**  **Enter heuristic values for each node:**  **Heuristic for C: 5**  **Heuristic for E: 3**  **Heuristic for A: 4**  **Heuristic for F: 2**  **Heuristic for B: 1**  **Heuristic for D: 0**  **Enter start node: A**  **Enter goal node: F**  **A\* Path: A -> B -> D -> E -> F** |
| **10) Implement A\* algorithm. Read undirected weighted graph and heuristic values from a .csv file.**  import pandas as pd  import heapq  def a\_star(graph, heuristics, start, goal):  open\_list = []  heapq.heappush(open\_list, (heuristics[start], start, [start], 0)) # (f, node, path, g)  visited = set()  while open\_list:  f, current, path, g = heapq.heappop(open\_list)  if current == goal:  print(f"A\* Path: {' -> '.join(path)}")  return  visited.add(current)  for neighbor, cost in graph.get(current, []):  if neighbor not in visited:  total\_cost = g + cost  heapq.heappush(open\_list, (total\_cost + heuristics[neighbor], neighbor, path + [neighbor], total\_cost))  print("Goal not reachable.")  # --- Reading CSV ---  edges\_df = pd.read\_csv("D:\------------------------------\SEM VI\AI LAB\Practical Exam\exp\_10\_a.csv")  heuristics\_df = pd.read\_csv("D:\------------------------------\SEM VI\AI LAB\Practical Exam\exp\_10\_heuristic.csv")  # Build graph  graph = {}  for \_, row in edges\_df.iterrows():  src, dest, cost = row['Source'], row['Destination'], row['Cost']  if src not in graph:  graph[src] = []  if dest not in graph:  graph[dest] = []  graph[src].append((dest, cost))  graph[dest].append((src, cost)) # because undirected  # Build heuristics  heuristics = {}  for \_, row in heuristics\_df.iterrows():  heuristics[row['Node']] = row['Heuristic']  # Input start and goal  start = input("Enter start node: ")  goal = input("Enter goal node: ")  # Run A\* Algorithm  a\_star(graph, heuristics, start, goal)  **Enter start node: A**  **Enter goal node: F**  **A\* Path: A -> B -> D -> E -> F** |
| **11) . Implement A\* algorithm. Read undirected weighted graph and heuristic values from user.**  **Enter number of edges: 5**  **Enter source node: A**  **Enter destination node: B**  **Enter cost from A to B: 1**  **Enter source node: A**  **Enter destination node: C**  **Enter cost from A to C: 4**  **Enter source node: B**  **Enter destination node: D**  **Enter cost from B to D: 2**  **Enter source node: C**  **Enter destination node: D**  **Enter cost from C to D: 1**  **Enter source node: D**  **Enter destination node: E**  **Enter cost from D to E: 5**  **Enter number of nodes for heuristic values: 6**  **Enter node: A**  **Enter heuristic value for A: 5**  **Enter node: B**  **Enter heuristic value for B: 3**  **Enter node: C**  **Enter heuristic value for C: 4**  **Enter node: D**  **Enter heuristic value for D: 2**  **Enter node: E**  **Enter heuristic value for E: 1**  **Enter node: F**  **Enter heuristic value for F: 0**  **Enter start node: A**  **Enter goal node: E** |
| **12) Implement Fuzzy set operations – union, intersection and complement. Demonstrate these operations with 3 fuzzy sets.**  def fuzzy\_union(set1, set2):  return {x: max(set1.get(x, 0), set2.get(x, 0)) for x in set(set1) | set(set2)}  def fuzzy\_intersection(set1, set2):  return {x: min(set1.get(x, 0), set2.get(x, 0)) for x in set(set1) & set(set2)}  def fuzzy\_complement(fuzzy\_set):  return {x: round(1 - v, 2) for x, v in fuzzy\_set.items()}  # Define three fuzzy sets  A = {'a': 0.2, 'b': 0.5, 'c': 0.7}  B = {'b': 0.6, 'c': 0.4, 'd': 0.9}  C = {'a': 0.8, 'd': 0.3, 'e': 0.5}  print("Fuzzy Set A:", A)  print("Fuzzy Set B:", B)  print("Fuzzy Set C:", C)  # Union of A and B  union\_AB = fuzzy\_union(A, B)  print("\nUnion of A and B:", union\_AB)  # Intersection of B and C  intersection\_BC = fuzzy\_intersection(B, C)  print("\nIntersection of B and C:", intersection\_BC)  # Complement of C  complement\_C = fuzzy\_complement(C)  print("\nComplement of C:", complement\_C)  **Fuzzy Set A: {'a': 0.2, 'b': 0.5, 'c': 0.7}**  **Fuzzy Set B: {'b': 0.6, 'c': 0.4, 'd': 0.9}**  **Fuzzy Set C: {'a': 0.8, 'd': 0.3, 'e': 0.5}**  **Union of A and B: {'a': 0.2, 'b': 0.6, 'd': 0.9, 'c': 0.7}**  **Intersection of B and C: {'d': 0.3}**  **Complement of C: {'a': 0.2, 'd': 0.7, 'e': 0.5}** |
| **13)** **Implement Fuzzy set operations – union, intersection and complement. Demonstrate De Morgan’s Law ( Complement of Union) with 2 fuzzy sets.**  **# 13th Experiment**  **# Fuzzy Set Operations - De Morgan's Law (Complement of Union)**  def fuzzy\_union(A, B):  return {x: max(A.get(x, 0), B.get(x, 0)) for x in set(A) | set(B)}  def fuzzy\_complement(A):  return {x: round(1 - val, 2) for x, val in A.items()}  # Input fuzzy sets  A = {}  B = {}  n = int(input("Enter number of elements in the fuzzy sets: "))  print("\nEnter elements for Set A:")  for \_ in range(n):  ele = input("Element: ")  val = float(input(f"Membership value of {ele} (between 0 and 1): "))  A[ele] = val  print("\nEnter elements for Set B:")  for \_ in range(n):  ele = input("Element: ")  val = float(input(f"Membership value of {ele} (between 0 and 1): "))  B[ele] = val  # Perform operations  union = fuzzy\_union(A, B)  complement\_of\_union = fuzzy\_complement(union)  complement\_A = fuzzy\_complement(A)  complement\_B = fuzzy\_complement(B)  intersection\_of\_complements = {x: min(complement\_A.get(x, 0), complement\_B.get(x, 0)) for x in set(complement\_A) | set(complement\_B)}  # Output  print("\nFuzzy Set A:", A)  print("Fuzzy Set B:", B)  print("\nUnion of A and B:", union)  print("Complement of (A ∪ B):", complement\_of\_union)  print("\nComplement of A:", complement\_A)  print("Complement of B:", complement\_B)  print("Intersection of (complement of A) and (complement of B):", intersection\_of\_complements)  # Verifying De Morgan's Law  if complement\_of\_union == intersection\_of\_complements:  print("\n De Morgan's Law Verified: Complement of (A ∪ B) == (Complement of A) ∩ (Complement of B)")  else:  print("\n De Morgan's Law NOT Verified")  **Enter number of elements in the fuzzy sets: 3**  **Enter elements for Set A:**  **Element: x**  **Membership value of x (between 0 and 1): 0.5**  **Element: y**  **Membership value of y (between 0 and 1): 0.8**  **Element: z**  **Membership value of z (between 0 and 1): 0.2**  **Enter elements for Set B:**  **Element: x**  **Membership value of x (between 0 and 1): 0.7**  **Element: y**  **Membership value of y (between 0 and 1): 0.4**  **Element: z**  **Membership value of z (between 0 and 1): 0.9**  **Fuzzy Set A: {'x': 0.5, 'y': 0.8, 'z': 0.2}**  **Fuzzy Set B: {'x': 0.7, 'y': 0.4, 'z': 0.9}**  **Union of A and B: {'y': 0.8, 'x': 0.7, 'z': 0.9}**  **Complement of (A ∪ B): {'y': 0.2, 'x': 0.3, 'z': 0.1}**  **Complement of A: {'x': 0.5, 'y': 0.2, 'z': 0.8}**  **Complement of B: {'x': 0.3, 'y': 0.6, 'z': 0.1}**  **Intersection of (complement of A) and (complement of B): {'y': 0.2, 'x': 0.3, 'z': 0.1}**  **✅ De Morgan's Law Verified: Complement of (A ∪ B) == (Complement of A) ∩ (Complement of B)** |
| **14) Implement Fuzzy set operations – union, intersection and complement. Demonstrate De Morgan’s Law ( Complement of Intersection) with 2 fuzzy sets.**  # 14th Experiment  # Fuzzy Set Operations - De Morgan's Law (Complement of Intersection)  def fuzzy\_intersection(A, B):  return {x: min(A.get(x, 0), B.get(x, 0)) for x in set(A) | set(B)}  def fuzzy\_complement(A):  return {x: round(1 - val, 2) for x, val in A.items()}  # Input fuzzy sets  A = {}  B = {}  n = int(input("Enter number of elements in the fuzzy sets: "))  print("\nEnter elements for Set A:")  for \_ in range(n):  ele = input("Element: ")  val = float(input(f"Membership value of {ele} (between 0 and 1): "))  A[ele] = val  print("\nEnter elements for Set B:")  for \_ in range(n):  ele = input("Element: ")  val = float(input(f"Membership value of {ele} (between 0 and 1): "))  B[ele] = val  # Perform operations  intersection = fuzzy\_intersection(A, B)  complement\_of\_intersection = fuzzy\_complement(intersection)  complement\_A = fuzzy\_complement(A)  complement\_B = fuzzy\_complement(B)  union\_of\_complements = {x: max(complement\_A.get(x, 0), complement\_B.get(x, 0)) for x in set(complement\_A) | set(complement\_B)}  # Output  print("\nFuzzy Set A:", A)  print("Fuzzy Set B:", B)  print("\nIntersection of A and B:", intersection)  print("Complement of (A ∩ B):", complement\_of\_intersection)  print("\nComplement of A:", complement\_A)  print("Complement of B:", complement\_B)  print("Union of (complement of A) and (complement of B):", union\_of\_complements)  # Verifying De Morgan's Law  if complement\_of\_intersection == union\_of\_complements:  print("\n✅ De Morgan's Law Verified: Complement of (A ∩ B) == (Complement of A) ∪ (Complement of B)")  else:  print("\n❌ De Morgan's Law NOT Verified")  **Enter number of elements in the fuzzy sets: 3**  **Enter elements for Set A:**  **Element: a**  **Membership value of a (between 0 and 1): 0.4**  **Element: b**  **Membership value of b (between 0 and 1): 0.9**  **Element: c**  **Membership value of c (between 0 and 1): 0.5**  **Enter elements for Set B:**  **Element: a**  **Membership value of a (between 0 and 1): 0.7**  **Element: b**  **Membership value of b (between 0 and 1): 0.3**  **Element: c**  **Membership value of c (between 0 and 1): 0.8**  **Fuzzy Set A: {'a': 0.4, 'b': 0.9, 'c': 0.5}**  **Fuzzy Set B: {'a': 0.7, 'b': 0.3, 'c': 0.8}**  **Intersection of A and B: {'a': 0.4, 'b': 0.3, 'c': 0.5}**  **Complement of (A ∩ B): {'a': 0.6, 'b': 0.7, 'c': 0.5}**  **Complement of A: {'a': 0.6, 'b': 0.1, 'c': 0.5}**  **Complement of B: {'a': 0.3, 'b': 0.7, 'c': 0.2}**  **Union of (complement of A) and (complement of B): {'a': 0.6, 'b': 0.7, 'c': 0.5}**  **✅ De Morgan's Law Verified: Complement of (A ∩ B) == (Complement of A) ∪ (Complement of B)** |
| 15)Tic-Tac-Toe  import random  # Constants for the board  PLAYER = 'X'  COMPUTER = 'O'  EMPTY = ' '  # A function to check if a player has won  def is\_winner(board, player):  win\_patterns = [  [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 pattern in win\_patterns:  if all([board[i] == player for i in pattern]):  return True  return False  # A function to check if the board is full (draw)  def is\_draw(board):  return all([space != EMPTY for space in board])  # Minimax Algorithm (Modified for the AI to win or draw)  def minimax(board, depth, is\_maximizing):  if is\_winner(board, COMPUTER):  return 1 # Computer wins  if is\_winner(board, PLAYER):  return -1 # Player wins  if is\_draw(board):  return 0 # Draw    if is\_maximizing:  best = -float('inf')  for i in range(9):  if board[i] == EMPTY:  board[i] = COMPUTER  best = max(best, minimax(board, depth + 1, False))  board[i] = EMPTY  return best  else:  best = float('inf')  for i in range(9):  if board[i] == EMPTY:  board[i] = PLAYER  best = min(best, minimax(board, depth + 1, True))  board[i] = EMPTY  return best  # Function to find the best move for the computer (AI)  def best\_move(board):  best\_val = -float('inf') # The computer is trying to win  best\_move = -1  for i in range(9):  if board[i] == EMPTY:  board[i] = COMPUTER  move\_val = minimax(board, 0, False)  board[i] = EMPTY  if move\_val > best\_val:  best\_move = i  best\_val = move\_val  return best\_move  # Function to print the board in a readable format  def print\_board(board):  for i in range(0, 9, 3):  print(board[i:i+3])  # Main Game Function  def play\_game():  board = [EMPTY] \* 9  print("Tic-Tac-Toe Game: You are 'X', Computer is 'O'")    while True:  print\_board(board)    # Player's turn (Human)  player\_move = int(input("Enter your move (1-9): ")) - 1  if board[player\_move] != EMPTY:  print("Invalid move! Try again.")  continue  board[player\_move] = PLAYER    if is\_winner(board, PLAYER):  print\_board(board)  print("You win!")  break  if is\_draw(board):  print\_board(board)  print("It's a draw!")  break    # Computer's turn  print("Computer's turn...")  move = best\_move(board)  board[move] = COMPUTER    if is\_winner(board, COMPUTER):  print\_board(board)  print("Computer wins!")  break  if is\_draw(board):  print\_board(board)  print("It's a draw!")  break  # Start the game  play\_game() |
| 16) Tic tac toe (computer loses or draws)  import random  # Constants for the board  PLAYER = 'X'  COMPUTER = 'O'  EMPTY = ' '  # A function to check if a player has won  def is\_winner(board, player):  win\_patterns = [  [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 pattern in win\_patterns:  if all([board[i] == player for i in pattern]):  return True  return False  # A function to check if the board is full (draw)  def is\_draw(board):  return all([space != EMPTY for space in board])  # Minimax Algorithm (Modified for the AI to lose or draw)  def minimax(board, depth, is\_maximizing):  if is\_winner(board, COMPUTER):  return 1 # Computer wins  if is\_winner(board, PLAYER):  return -1 # Player wins  if is\_draw(board):  return 0 # Draw    if is\_maximizing:  best = float('inf')  for i in range(9):  if board[i] == EMPTY:  board[i] = COMPUTER  best = min(best, minimax(board, depth + 1, False))  board[i] = EMPTY  return best  else:  best = -float('inf')  for i in range(9):  if board[i] == EMPTY:  board[i] = PLAYER  best = max(best, minimax(board, depth + 1, True))  board[i] = EMPTY  return best  # Function to find the best move for the computer (AI)  def best\_move(board):  best\_val = float('inf') # The computer is trying to lose  best\_move = -1  for i in range(9):  if board[i] == EMPTY:  board[i] = COMPUTER  move\_val = minimax(board, 0, False)  board[i] = EMPTY  if move\_val < best\_val:  best\_move = i  best\_val = move\_val  return best\_move  # Function to print the board in a readable format  def print\_board(board):  for i in range(0, 9, 3):  print(board[i:i+3])  # Main Game Function  def play\_game():  board = [EMPTY] \* 9  print("Tic-Tac-Toe Game: You are 'X', Computer is 'O'")    while True:  print\_board(board)    # Player's turn (Human)  player\_move = int(input("Enter your move (1-9): ")) - 1  if board[player\_move] != EMPTY:  print("Invalid move! Try again.")  continue  board[player\_move] = PLAYER    if is\_winner(board, PLAYER):  print\_board(board)  print("You win!")  break  if is\_draw(board):  print\_board(board)  print("It's a draw!")  break    # Computer's turn  print("Computer's turn...")  move = best\_move(board)  board[move] = COMPUTER    if is\_winner(board, COMPUTER):  print\_board(board)  print("Computer wins!")  break  if is\_draw(board):  print\_board(board)  print("It's a draw!")  break  # Start the game  play\_game() |

|  |
| --- |
| **17) Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one binary output. Display the final weight matrices, bias values and the number of steps. Note that random values are assigned to weight matrices and bias in each step.**  **import numpy as np**  # Number of binary inputs (you can change this)  N = 3  # Number of neurons in hidden layers  hidden\_layer1\_size = 4  hidden\_layer2\_size = 3  # Activation function: Step function (binary output)  def activation(x):  return np.where(x >= 0.5, 1, 0)  # Generate random binary input  inputs = np.random.randint(0, 2, (1, N))  print("Input:", inputs)  # Step counter  steps = 0  while True:  steps += 1  # Randomly initialize weights and biases  W1 = np.random.rand(N, hidden\_layer1\_size)  b1 = np.random.rand(1, hidden\_layer1\_size)    W2 = np.random.rand(hidden\_layer1\_size, hidden\_layer2\_size)  b2 = np.random.rand(1, hidden\_layer2\_size)    W3 = np.random.rand(hidden\_layer2\_size, 1)  b3 = np.random.rand(1, 1)  # Forward pass  z1 = np.dot(inputs, W1) + b1  a1 = activation(z1)    z2 = np.dot(a1, W2) + b2  a2 = activation(z2)    z3 = np.dot(a2, W3) + b3  output = activation(z3)    # Check if output is 1 (you can change criteria)  if output[0][0] == 1:  break  # Display results  print("\nFinal Output:", output)  print("\nFinal Weight matrices and Biases:")  print("W1 (Input -> Hidden Layer 1):\n", W1)  print("b1 (Bias for Hidden Layer 1):\n", b1)  print("\nW2 (Hidden Layer 1 -> Hidden Layer 2):\n", W2)  print("b2 (Bias for Hidden Layer 2):\n", b2)  print("\nW3 (Hidden Layer 2 -> Output):\n", W3)  print("b3 (Bias for Output Layer):\n", b3)  print("\nNumber of steps (iterations):", steps)  **✨ Explanation:**   * **Inputs: Random binary (0 or 1) input vector of size N.** * **Weights and Biases: Randomized in each step.** * **Activation: Simple binary step function (output 0 or 1).** * **Stopping Condition:**   + **We stop when the network outputs 1.**   + **You can modify the stopping condition based on your requirement.** * **After stopping: Print**   + **Final weights (W1, W2, W3)**   + **Final biases (b1, b2, b3)**   + **Number of steps taken.** |
| **18) Implement a simple Multi-Layer Perceptron with 4 binary inputs, one hidden layer and two binary outputs. Display the final weight matrices, bias values and the number of steps. Note that random values are assigned to weight matrices and bias in each step.**  import numpy as np  # Activation function: Sigmoid  def sigmoid(x):  return 1 / (1 + np.exp(-x))  # Derivative of sigmoid  def sigmoid\_derivative(x):  return x \* (1 - x)  # Input dataset (4 binary inputs)  X = np.array([  [0, 0, 0, 0],  [0, 0, 1, 1],  [1, 1, 0, 0],  [1, 1, 1, 1]  ])  # Output dataset (2 binary outputs)  y = np.array([  [0, 0],  [1, 0],  [0, 1],  [1, 1]  ])  # Seed for reproducibility  np.random.seed(42)  # Initialize weights and biases randomly  input\_neurons = 4  hidden\_neurons = 5  output\_neurons = 2  # Weights  W1 = np.random.rand(input\_neurons, hidden\_neurons)  W2 = np.random.rand(hidden\_neurons, output\_neurons)  # Biases  b1 = np.random.rand(1, hidden\_neurons)  b2 = np.random.rand(1, output\_neurons)  # Learning rate  lr = 0.5  # Training loop  steps = 0  for epoch in range(10000): # Maximum 10000 steps  steps += 1    # Forward propagation  hidden\_input = np.dot(X, W1) + b1  hidden\_output = sigmoid(hidden\_input)  final\_input = np.dot(hidden\_output, W2) + b2  final\_output = sigmoid(final\_input)  # Calculate error  error = y - final\_output  if np.all(np.abs(error) < 0.01):  break # Stop if the error is very small (converged)  # Backpropagation  d\_final\_output = error \* sigmoid\_derivative(final\_output)  error\_hidden\_layer = d\_final\_output.dot(W2.T)  d\_hidden\_layer = error\_hidden\_layer \* sigmoid\_derivative(hidden\_output)  # Update weights and biases  W2 += hidden\_output.T.dot(d\_final\_output) \* lr  b2 += np.sum(d\_final\_output, axis=0, keepdims=True) \* lr  W1 += X.T.dot(d\_hidden\_layer) \* lr  b1 += np.sum(d\_hidden\_layer, axis=0, keepdims=True) \* lr  # Final Results  print("\nFinal Weights between Input and Hidden layer (W1):\n", W1)  print("\nFinal Biases for Hidden layer (b1):\n", b1)  print("\nFinal Weights between Hidden and Output layer (W2):\n", W2)  print("\nFinal Biases for Output layer (b2):\n", b2)  print("\nTotal Steps Taken:", steps)  **Output explanation:**   * **Final Weights (W1 and W2):**   + **W1: Matrix of weights between the input layer and the hidden layer.**   + **W2: Matrix of weights between the hidden layer and the output layer.**   + **After training, these values are adjusted so that the MLP gives correct output.** * **Final Biases (b1 and b2):**   + **b1: Biases for each neuron in the hidden layer.**   + **b2: Biases for each neuron in the output layer.**   + **Biases shift the activation function curve to help in better learning.** * **Total Steps Taken:**   + **Number of iterations (steps) the model took to learn the correct mapping.**   + **Training stops early if the prediction error becomes very small (near zero).** |
| **19) Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and Sigmoid function as activation function.**  import numpy as np  # Sigmoid activation and its derivative  def sigmoid(x):  return 1 / (1 + np.exp(-x))  def sigmoid\_derivative(x):  return x \* (1 - x)  # Define the MLP class  class MLP:  def \_\_init\_\_(self, input\_size, hidden1\_size, hidden2\_size, output\_size, learning\_rate=0.5):  # Randomly initialize weights and biases  self.W1 = np.random.rand(input\_size, hidden1\_size)  self.b1 = np.random.rand(hidden1\_size)  self.W2 = np.random.rand(hidden1\_size, hidden2\_size)  self.b2 = np.random.rand(hidden2\_size)  self.W3 = np.random.rand(hidden2\_size, output\_size)  self.b3 = np.random.rand(output\_size)  self.learning\_rate = learning\_rate  def forward(self, X):  self.z1 = sigmoid(np.dot(X, self.W1) + self.b1)  self.z2 = sigmoid(np.dot(self.z1, self.W2) + self.b2)  self.output = sigmoid(np.dot(self.z2, self.W3) + self.b3)  return self.output  def backward(self, X, y):  error = y - self.output  d\_output = error \* sigmoid\_derivative(self.output)  error\_hidden2 = d\_output.dot(self.W3.T)  d\_hidden2 = error\_hidden2 \* sigmoid\_derivative(self.z2)  error\_hidden1 = d\_hidden2.dot(self.W2.T)  d\_hidden1 = error\_hidden1 \* sigmoid\_derivative(self.z1)  # Update weights and biases  self.W3 += self.z2.T.dot(d\_output) \* self.learning\_rate  self.b3 += np.sum(d\_output, axis=0) \* self.learning\_rate  self.W2 += self.z1.T.dot(d\_hidden2) \* self.learning\_rate  self.b2 += np.sum(d\_hidden2, axis=0) \* self.learning\_rate  self.W1 += X.T.dot(d\_hidden1) \* self.learning\_rate  self.b1 += np.sum(d\_hidden1, axis=0) \* self.learning\_rate  def train(self, X, y, epochs=10000):  for step in range(epochs):  output = self.forward(X)  self.backward(X, y)  if step % 1000 == 0:  loss = np.mean((y - output) \*\* 2)  print(f"Step {step} Loss: {loss:.4f}")  print("\nTraining completed.")  print("\nFinal Weights and Biases:")  print("W1:", self.W1)  print("b1:", self.b1)  print("W2:", self.W2)  print("b2:", self.b2)  print("W3:", self.W3)  print("b3:", self.b3)  # Sample Usage  if \_\_name\_\_ == "\_\_main\_\_":  # Binary input (4 examples, N=4 inputs)  X = np.array([  [0, 0, 0, 0],  [0, 0, 1, 1],  [1, 1, 0, 0],  [1, 1, 1, 1]  ])  # Output (Binary)  y = np.array([[0], [1], [1], [0]])  mlp = MLP(input\_size=4, hidden1\_size=5, hidden2\_size=3, output\_size=1)  mlp.train(X, y)  **Step 0 Loss: 0.3635**  **Step 1000 Loss: 0.1725**  **Step 2000 Loss: 0.0069**  **Step 3000 Loss: 0.0013**  **Step 4000 Loss: 0.0007**  **Step 5000 Loss: 0.0005**  **Step 6000 Loss: 0.0003**  **Step 7000 Loss: 0.0003**  **Step 8000 Loss: 0.0002**  **Step 9000 Loss: 0.0002**  **Training completed.**  **Final Weights and Biases:**  **W1: [[1.76086764 2.73133904 1.47709118 0.75935121 1.5136256 ]**  **[1.69758405 2.95642642 1.29571867 1.29983526 1.33015977]**  **[1.57134571 2.66569303 1.19460491 1.27815937 1.76160978]**  **[1.99077663 3.0138101 1.60765043 1.23044898 1.07496946]]**  **b1: [-1.31033645 -2.44614295 -4.26450894 -0.28331364 -4.24912598]**  **W2: [[ 2.31858811 -0.74800702 -0.6186628 ]**  **[ 5.14092311 -2.71907583 -0.45141492]**  **[-0.95953637 4.89162766 1.44860252]**  **[ 0.62030263 -0.2562238 0.12646584]**  **[ 2.19541255 4.2505465 1.48172753]]**  **b2: [-3.82556333 -0.80919613 -0.71275592]**  **W3: [[ 7.98590969]**  **[-9.00981411]**  **[-2.16064005]]**  **b3: [-2.28936441]** |
| **20) Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and ReLU function as activation function.**  import numpy as np  # ReLU activation function and its derivative  def relu(x):  return np.maximum(0, x)  def relu\_derivative(x):  return np.where(x > 0, 1, 0)  # Define the MLP class  class MLP:  def \_\_init\_\_(self, input\_size, hidden1\_size, hidden2\_size, output\_size, learning\_rate=0.01):  self.W1 = np.random.randn(input\_size, hidden1\_size)  self.b1 = np.random.randn(hidden1\_size)  self.W2 = np.random.randn(hidden1\_size, hidden2\_size)  self.b2 = np.random.randn(hidden2\_size)  self.W3 = np.random.randn(hidden2\_size, output\_size)  self.b3 = np.random.randn(output\_size)  self.learning\_rate = learning\_rate  def forward(self, X):  self.z1 = relu(np.dot(X, self.W1) + self.b1)  self.z2 = relu(np.dot(self.z1, self.W2) + self.b2)  self.output = relu(np.dot(self.z2, self.W3) + self.b3)  return self.output  def backward(self, X, y):  error = y - self.output  d\_output = error \* relu\_derivative(self.output)  error\_hidden2 = d\_output.dot(self.W3.T)  d\_hidden2 = error\_hidden2 \* relu\_derivative(self.z2)  error\_hidden1 = d\_hidden2.dot(self.W2.T)  d\_hidden1 = error\_hidden1 \* relu\_derivative(self.z1)  # Update weights and biases  self.W3 += self.z2.T.dot(d\_output) \* self.learning\_rate  self.b3 += np.sum(d\_output, axis=0) \* self.learning\_rate  self.W2 += self.z1.T.dot(d\_hidden2) \* self.learning\_rate  self.b2 += np.sum(d\_hidden2, axis=0) \* self.learning\_rate  self.W1 += X.T.dot(d\_hidden1) \* self.learning\_rate  self.b1 += np.sum(d\_hidden1, axis=0) \* self.learning\_rate  def train(self, X, y, epochs=10000):  for step in range(epochs):  output = self.forward(X)  self.backward(X, y)  if step % 1000 == 0:  loss = np.mean((y - output) \*\* 2)  print(f"Step {step} Loss: {loss:.4f}")  print("\nTraining completed.")  print("\nFinal Weights and Biases:")  print("W1:", self.W1)  print("b1:", self.b1)  print("W2:", self.W2)  print("b2:", self.b2)  print("W3:", self.W3)  print("b3:", self.b3)  # Sample Usage  if \_\_name\_\_ == "\_\_main\_\_":  # Binary input (4 examples, N=4 inputs)  X = np.array([  [0, 0, 0, 0],  [0, 0, 1, 1],  [1, 1, 0, 0],  [1, 1, 1, 1]  ])  # Output (Binary)  y = np.array([[0], [1], [1], [0]])  mlp = MLP(input\_size=4, hidden1\_size=5, hidden2\_size=3, output\_size=1)  mlp.train(X, y)  **Step 0 Loss: 0.3139**  **Step 1000 Loss: 0.2500**  **Step 2000 Loss: 0.2500**  **Step 3000 Loss: 0.2500**  **Step 4000 Loss: 0.2500**  **Step 5000 Loss: 0.2500**  **Step 6000 Loss: 0.2500**  **Step 7000 Loss: 0.2500**  **Step 8000 Loss: 0.2500**  **Step 9000 Loss: 0.2500**  **Training completed.**  **Final Weights and Biases:**  **W1: [[-1.10633497 -1.19620662 0.81252582 1.35624003 -0.07201012]**  **[ 1.0035329 0.36163603 -0.64511975 0.36139561 1.53803657]**  **[-0.03582604 1.56464366 -2.6197451 0.8219025 0.08704707]**  **[-0.29900735 0.09176078 -1.98756891 -0.21967189 0.35711257]]**  **b1: [ 1.45467532 -0.51827022 -0.8084936 -0.50175704 0.93690508]**  **W2: [[ 0.32875111 -0.5297602 0.442058 ]**  **[ 0.09707755 0.96864499 -0.70205309]**  **[-0.32766215 -0.39210815 -1.46351495]**  **[ 0.29612028 0.26105527 0.00511346]**  **[-0.23458713 -1.41537074 -0.46561146]]**  **b2: [-0.34271452 -0.80227727 -0.2098545 ]**  **W3: [[0.40405086]**  **[1.8861859 ]**  **[0.14217757]]**  **b3: [0.5]**  **Short Output Explanation:**  **✅ After training, the code displays:**   * **The final weight matrices (W1, W2, W3)** * **The final bias values (b1, b2, b3)** * **Training loss at every 1000 steps to show improvement.** * **At the end, you can see how the network learned to map the binary inputs to the correct output.** |
| **21) Implement a simple Multi-Layer Perceptron with N binary inputs, two hidden layers and one output. Use backpropagation and Tanh function as activation function.**  import numpy as np  # Activation function: Tanh and its derivative  def tanh(x):  return np.tanh(x)  def tanh\_derivative(x):  return 1 - np.tanh(x) \*\* 2  # Generate random binary input and output  np.random.seed(0)  N = 4 # Number of input features  X = np.random.randint(0, 2, (10, N)) # 10 samples  y = np.random.randint(0, 2, (10, 1)) # 10 output labels (binary)  # Initialize weights and biases  W1 = np.random.randn(N, 5)  b1 = np.random.randn(1, 5)  W2 = np.random.randn(5, 3)  b2 = np.random.randn(1, 3)  W3 = np.random.randn(3, 1)  b3 = np.random.randn(1, 1)  # Training parameters  learning\_rate = 0.1  epochs = 5000  # Training loop  for epoch in range(epochs):  # Forward pass  z1 = np.dot(X, W1) + b1  a1 = tanh(z1)  z2 = np.dot(a1, W2) + b2  a2 = tanh(z2)  z3 = np.dot(a2, W3) + b3  output = tanh(z3)  # Loss (Mean Squared Error)  loss = np.mean((output - y) \*\* 2)  # Backward pass  d\_loss\_output = 2 \* (output - y) / y.size  d\_output\_z3 = tanh\_derivative(z3)  d\_loss\_z3 = d\_loss\_output \* d\_output\_z3  d\_loss\_W3 = np.dot(a2.T, d\_loss\_z3)  d\_loss\_b3 = np.sum(d\_loss\_z3, axis=0, keepdims=True)  d\_loss\_a2 = np.dot(d\_loss\_z3, W3.T)  d\_a2\_z2 = tanh\_derivative(z2)  d\_loss\_z2 = d\_loss\_a2 \* d\_a2\_z2  d\_loss\_W2 = np.dot(a1.T, d\_loss\_z2)  d\_loss\_b2 = np.sum(d\_loss\_z2, axis=0, keepdims=True)  d\_loss\_a1 = np.dot(d\_loss\_z2, W2.T)  d\_a1\_z1 = tanh\_derivative(z1)  d\_loss\_z1 = d\_loss\_a1 \* d\_a1\_z1  d\_loss\_W1 = np.dot(X.T, d\_loss\_z1)  d\_loss\_b1 = np.sum(d\_loss\_z1, axis=0, keepdims=True)  # Update weights and biases  W3 -= learning\_rate \* d\_loss\_W3  b3 -= learning\_rate \* d\_loss\_b3  W2 -= learning\_rate \* d\_loss\_W2  b2 -= learning\_rate \* d\_loss\_b2  W1 -= learning\_rate \* d\_loss\_W1  b1 -= learning\_rate \* d\_loss\_b1  if epoch % 1000 == 0:  print(f"Epoch {epoch}, Loss: {loss:.4f}")  # Final results  print("\nFinal Weights and Biases:")  print("W1:", W1)  print("b1:", b1)  print("W2:", W2)  print("b2:", b2)  print("W3:", W3)  print("b3:", b3)  **Epoch 0, Loss: 2.2997**  **Epoch 1000, Loss: 0.0673**  **Epoch 2000, Loss: 0.0669**  **Epoch 3000, Loss: 0.0668**  **Epoch 4000, Loss: 0.0668**  **Final Weights and Biases:**  **W1: [[-1.08705076 0.36573398 0.06491499 0.68232188 -1.56214933]**  **[-0.91872066 -0.07991952 1.19047182 -0.72228004 -0.05183369]**  **[ 1.33894597 1.52276314 0.46669844 0.04934617 -1.11072183]**  **[ 1.43524447 1.78769591 -1.33320149 0.55427249 -0.4974763 ]]**  **b1: [[-1.18680524 0.80563212 -1.17544968 -1.04584274 0.91676422]]**  **W2: [[-1.97986637 -0.61001187 0.39902952]**  **[-0.18732698 -0.06596553 0.33933073]**  **[-0.88091333 -0.53321603 -2.18779099]**  **[-0.69179512 -1.28066949 0.21354917]**  **[-0.7176664 0.53457388 0.31458757]]**  **b2: [[ 1.07398084 -0.03685228 0.51483771]]**  **W3: [[0.07702197]**  **[2.31335219]**  **[1.5670253 ]]**  **b3: [[-0.23125367]]**  ** We implemented a Multi-Layer Perceptron (MLP) with N binary inputs, two hidden layers, and one output.**  ** Tanh is used as the activation function in all layers.**  ** Backpropagation is applied to train the network — it adjusts the weights and biases to reduce error.**  ** Functions used: tanh(x) and its derivative tanh\_derivative(x).**  ** The network trains using forward pass → loss calculation → backward pass → weight updates.**  ** After training, final weights, biases, and number of steps (epochs) are displayed.** |
| 22) NOT WORKING  **Write a program to read a text file with at least 30 sentences and 200 words and perform the following tasks in the given sequence. a. b. c. d. e. 23. Text cleaning by removing punctuation/special characters, numbers and extra white spaces. Use regular expression for the same. Convert text to lowercase Tokenization Remove stop words Correct misspelled words**  import nltk  # Download the required resources  nltk.download('punkt')  nltk.download('stopwords')  # Re-import the necessary modules after downloading  from nltk.tokenize import word\_tokenize  from nltk.corpus import stopwords  from spellchecker import SpellChecker  import re  # Step 1: Read file  def read\_file(file\_path):  with open(file\_path, 'r') as file:  return file.read()  # Step 2: Clean the text  def clean\_text(text):  # Remove punctuation, special characters, and numbers using regex  text = re.sub(r'[^a-zA-Z\s]', '', text)  # Remove extra whitespaces  text = re.sub(r'\s+', ' ', text)  return text  # Step 3: Convert text to lowercase  def to\_lowercase(text):  return text.lower()  # Step 4: Tokenization  def tokenize(text):  return word\_tokenize(text)  # Step 5: Remove stop words  def remove\_stopwords(tokens):  stop\_words = set(stopwords.words('english'))  return [word for word in tokens if word not in stop\_words]  # Step 6: Correct misspelled words  def correct\_spelling(tokens):  spell = SpellChecker()  return [spell.correction(word) for word in tokens]  # Main function to process text  def process\_text(file\_path):  # Step 1: Read file  text = read\_file(file\_path)    # Step 2: Clean the text  text = clean\_text(text)    # Step 3: Convert text to lowercase  text = to\_lowercase(text)    # Step 4: Tokenization  tokens = tokenize(text)    # Step 5: Remove stop words  tokens = remove\_stopwords(tokens)    # Step 6: Correct misspelled words  tokens = correct\_spelling(tokens)    return tokens  # Example usage  file\_path = "/content/exp\_22.txt" # Replace with the path to your text file  processed\_tokens = process\_text(file\_path)  print("Processed Tokens: ", processed\_tokens) |
|  |
|  |