PANDIT DEENDAYAL ENERGY UNIVERSITY

Raysan, Gandhinagar –382426,Gujarat,

India

Artificial Intelligence Lab

23CP307P

Name - Navghan Kabira

Roll Number: 21BCP327

Group - G10

Division – 5

Practical 1

Aim: WAP to implement DFS and BFS for traversing a graph from source node (S) to goal node (G), where source node and goal node is given by the user as an input.

Theory:

Depth-First Search (DFS) and Breadth-First Search (BFS) are two fundamental graph traversal algorithms used to visit all the nodes in a graph. Here's a brief overview of each:

1. **Depth-First Search (DFS):**
   - DFS explores as far as possible along each branch before backtracking.
   - It uses a stack data structure (or recursion) to keep track of nodes to visit next.
   - DFS is often implemented using recursion.
   - It traverses the graph in a depthward motion.
2. Breadth-First Search (BFS):
   - BFS explores neighbor nodes before moving to the next level of nodes.
   - It uses a queue data structure to keep track of nodes to visit next.
   - BFS is suitable for finding the shortest path in an unweighted graph.
   - It traverses the graph in a breadthward motion.

Code:

```
from collections import defaultdict

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def dfs_util(self, current, goal, visited):
        if current == goal:
            return True
        visited.add(current)
```

```
        for neighbor in self.graph[current]:
            if neighbor not in visited:
                if self.dfs_util(neighbor, goal, visited):
                    return True
        return False
    def dfs(self, start, goal):
        visited = set()
        return self.dfs_util(start, goal, visited)
    def bfs(self, start, goal):
        visited = set()
        queue = [start]
        while queue:
            current_node = queue.pop(0)
            if current_node == goal:
                return True
            visited.add(current_node)

            for neighbor in self.graph[current_node]:
                if neighbor not in visited:
                    queue.append(neighbor)

        return False

# Create an instance of the Graph class
graph = Graph()

# Take edges as input from the user
num_edges = int(input("Enter the number of edges: "))
for _ in range(num_edges):
    edge = input("Enter an edge (format: u v): ").split()
    graph.add_edge(edge[0], edge[1])

# Take source and goal nodes as input from the user
source_node = input('Enter the source node: ')
goal_node = input('Enter the goal node: ')

# Check for path using DFS and print the result
if graph.dfs(source_node, goal_node):
    print('DFS: True')
else:
```

```
    print('DFS: False')

# Check for path using BFS and print the result
if graph.bfs(source_node, goal_node):
    print('BFS: True')
else:
    print('BFS: False')
```

Output:

```
Enter the number of edges: 7
Enter an edge (format: u v): S A
Enter an edge (format: u v): S B
Enter an edge (format: u v): A C
Enter an edge (format: u v): B D
Enter an edge (format: u v): B E
Enter an edge (format: u v): D G
Enter an edge (format: u v): E G
Enter the source node: S
Enter the goal node: G
DFS: True
BFS: True
```

Conclusion:

In this practical, we implemented DFS and BFS algorithms to traverse a graph from a given source node to a goal node. The implementation allows the user to input the edges of the graph, as well as the source and goal nodes. The algorithms then search for a path from the source to the goal using both DFS and BFS methods.

Practical 2

Aim: Design waterjug problem solver

Problem Statement :

You are given two jugs with m litres and a n litre capacity. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n. You are given two jugs with m litres and a n litre capacity. Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n.

Theory :

- Water jug problem using BFS or DFS:

1. Start with an Empty State: (0, 0)

   Both jugs are initially empty.

2. Apply All Possible Actions from the Current State: (0, 0)

   the 3-liter jug: (3, 0)

   Fill the 5-liter jug: (0, 5)

3. Expand to the Next Level:

   We now have two new states to explore, (3, 0) and (0, 5).

4. Continue Expanding:

   From (3, 0), we can:

   Pour from the 3-liter jug to the 5-liter jug: (0, 3)

   Fill the 3-liter jug again: (3, 3)

   From (0, 5), we can:

   Pour from the 5-liter jug to the 3-liter jug: (3, 2)

   Empty the 5-liter jug: (0, 0)

5. Explore Further:

   Continue expanding the states at the next level:

   From (0, 3), we can pour to reach (3, 0).

From (3, 3), we can reach (0, 3) or (3, 5).

6. Goal State Achieved:

In our search, we've reached the goal state (0, 4).

7. Backtrack to Find the Solution:

To find the solution path, we backtrack from the goal state to the initial state:

(0, 4) -> (3, 1) -> (0, 1) -> (1, 0) -> (1, 5) -> (3, 4) -> (0, 4)

- This step-by-step demonstration shows how Breadth-First Search systematically explores the state space to find the optimal solution to the Water Jug Problem.

Code:

```
from collections import deque
def water_jug_BFS(j1, j2, target):
    visited = set()
    queue = deque([(0, 0, [])])  # Adding touple which contain first
step as (0,0) and following step in list form
    while queue:
        jug_a, jug_b, sequence = queue.popleft()

        if jug_a == target or jug_b == target:
            return True, sequence  # Return true and sequence which
contain steps when targeted amount got

        if (jug_a, jug_b) in visited:
            continue

        visited.add((jug_a, jug_b))    #(jug_a,jug_b) in visited then
skip it

        # Filling jug A
        if jug_a < j1:
            new_state = (j1, jug_b)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Fill jug A', new_state)]))

        # Filling jug B
        if jug_b < j2:
            new_state = (jug_a, j2)
```

```
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Fill jug B', new_state)]))

        # Emptying jug A
        if jug_a > 0:
            new_state = (0, jug_b)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Empty jug A', new_state)]))

        # Emptying jug B
        if jug_b > 0:
            new_state = (jug_a, 0)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Empty jug B', new_state)]))

        # Pouring from A to B
        if jug_a + jug_b >= j2:
            new_state = (jug_a - (j2 - jug_b), j2)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Pour from A to B', new_state)]))
        else:
            new_state = (0, jug_a + jug_b)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Pour from A to B', new_state)]))

        # Pouring from B to A
        if jug_a + jug_b >= j1:
            new_state = (j1, jug_b - (j1 - jug_a))
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Pour from B to A', new_state)]))
        else:
            new_state = (jug_a + jug_b, 0)
            queue.append((new_state[0], new_state[1], sequence +
[((jug_a, jug_b), 'Pour from B to A', new_state)]))

    return False, []    #if solution not possible it return false and
empty tuple


j1 = int(input('Enter the Capacity of Jug A : '))  # Capacity of jug A
```

```
j2 = int(input('Enter the Capacity of Jug B : '))  # Capacity of jug B
target = int(input('Enter the Amount of Water : '))  # Desired amount
of water
possible, sequence = water_jug_BFS(j1, j2, target)

if possible:
    print('Sequence of steps : ')
    for state, step, new_state in sequence:
        print(f'{state} -> {step} -> {new_state}')
else:
    print(f'Problem is not solvable')
```

Output:

```
Enter the Capacity of Jug A : 5
Enter the Capacity of Jug B : 6
Enter the Amount of Water : 2
Sequence of steps :
(0, 0) -> Fill jug B -> (0, 6)
(0, 6) -> Pour from B to A -> (5, 1)
(5, 1) -> Empty jug A -> (0, 1)
(0, 1) -> Pour from B to A -> (1, 0)
(1, 0) -> Fill jug B -> (1, 6)
(1, 6) -> Pour from B to A -> (5, 2)
```

Conclusion:

In this practical, we implemented a solution to the Water Jug Problem using the BFS algorithm. The program takes inputs for the capacities of Jug A and Jug B, as well as the desired amount of water to measure. It then uses BFS to find a sequence of steps to achieve the desired quantity of water. If a solution is found, the program prints the sequence of steps. Otherwise, it indicates that the problem is not solvable.

Practical 3

Aim: Solve 8 puzzle problem using A* algorithm where initial state and Goal state will be given by the users.

Theory:

1. Problem Description:

- The 8 Puzzle Problem involves a 3x3 grid containing numbered tiles, with one tile left blank. The goal is to rearrange the tiles from an initial configuration to a goal configuration by sliding them into the blank space.

2. A* Algorithm:

- A* is a search algorithm used to find the shortest path from a start node to a goal node in a graph or state space. It combines the benefits of uniform-cost search and greedy best-first search.

3. Heuristic Function:

- A* relies on a heuristic function that estimates the cost from the current state to the goal state. In the 8 Puzzle Problem, common heuristics include misplaced tiles or Manhattan distance.

4. State Representation:

- Each state of the puzzle is represented as a node in the search space, containing the current configuration of the puzzle, the cost to reach that state, the heuristic estimate of the remaining cost to the goal, and the total cost.

5. Search Process:

- A* starts with an initial state and explores successor states by applying valid moves (swapping the blank tile with adjacent tiles). It calculates the cost, heuristic, and total cost for each successor state and prioritizes states with lower total costs in the search.

6. Goal Test:

- The search terminates when the goal state is reached, indicating that a solution has been found. If the priority queue becomes empty before reaching the goal, it indicates that no solution is found.

7. Solution Path:

- Once a solution is found, the algorithm outputs the sequence of steps needed to reach the goal state from the initial state. These steps represent the shortest path through the state space.

Code:

```
import heapq
N = 3

class State:
    def __init__(self, puzzle, cost, heuristic, totalCost):
        self.puzzle = puzzle
        self.cost = cost
        self.heuristic = heuristic
        self.totalCost = totalCost

    def __lt__(self, other):
        return self.heuristic < other.heuristic

def printPuzzle(puzzle):
    for i in range(N):
        for j in range(N):
            print(puzzle[i][j], end=" ")
        print()

def findNumber(puzzle, num):
    for i in range(N):
        for j in range(N):
            if puzzle[i][j] == num:
                return (i, j)
    return (-1, -1)

def misplacedTiles(current, goal):
    count = 0
    for i in range(N):
        for j in range(N):
            if current[i][j] != goal[i][j] and current[i][j] != 0:
                count += 1
    return count

def isGoalState(state, goal):
```

```
    return state == goal

def generateMoves(currentState, goal):
    moves = []
    xdir = [0, 0, 1, -1]
    ydir = [1, -1, 0, 0]

    for i in range(4):
        x, y = findNumber(currentState.puzzle, 0)
        newx = x + xdir[i]
        newy = y + ydir[i]

        if 0 <= newx < N and 0 <= newy < N:
            newPuzzle = [row[:] for row in currentState.puzzle]
            newPuzzle[x][y], newPuzzle[newx][newy] =
newPuzzle[newx][newy], newPuzzle[x][y]
            newState = State(newPuzzle, currentState.cost + 1,
misplacedTiles(newPuzzle, goal), currentState.cost + 1 +
misplacedTiles(newPuzzle, goal))
            moves.append(newState)
    return moves

def aStar(initial, goal):
    pq = []
    heapq.heappush(pq, initial)
    visited = set()
    steps = 0
    maxSteps = 20
    while pq:
        currentState = heapq.heappop(pq)
        print("Step", currentState.cost, ":")
        printPuzzle(currentState.puzzle)

        if isGoalState(currentState.puzzle, goal):
            print("Solution found!")
            print("Total Cost:", currentState.cost)
            return

        moves = generateMoves(currentState, goal)
        for move in moves:
```

```
            if tuple(map(tuple, move.puzzle)) not in visited:
                heapq.heappush(pq, move)
                visited.add(tuple(map(tuple, move.puzzle)))
        steps += 1

        if steps > maxSteps:
            print("No solution found within the specified number of
steps.")
            return
    print("No solution found.")
def main():
    print("Enter the initial state as a 3x3 matrix (0 represents the
blank tile):")
    initial = [list(map(int, input().split())) for _ in range(3)]
    print("Enter the goal state as a 3x3 matrix:")
    goal = [list(map(int, input().split())) for _ in range(3)]
    print("Initial state:")
    printPuzzle(initial)
    print("Goal state:")
    printPuzzle(goal)
    # Solve the puzzle using A*
    aStar(State(initial, 0, misplacedTiles(initial, goal),
misplacedTiles(initial, goal)), goal)
if __name__ == "__main__":
    main()
```

Output:

```
Enter the initial state as a 3x3 matrix (0 represents the blank tile):
1 2 3
4 5 6
7 8 0
Enter the goal state as a 3x3 matrix:
1 2 3
7 4 6
0 5 8
Initial state:
1 2 3
4 5 6
7 8 0
Goal state:
1 2 3
7 4 6
0 5 8
Step 0 :
```

```
Step 0 :
1 2 3
4 5 6
7 8 0
Step 1 :
1 2 3
4 5 6
7 0 8
Step 2 :
1 2 3
4 0 6
7 5 8
Step 3 :
1 2 3
0 4 6
7 5 8
Step 4 :
1 2 3
7 4 6
0 5 8
Solution found!
Total Cost: 4
```

Conclusion:

In this practical, we implemented the A* algorithm to solve the 8 Puzzle Problem. The program takes inputs for the initial and goal states of the puzzle from the user and then applies the A* algorithm to find the shortest path from the initial state to the goal state. If a solution is found, it prints the sequence of steps needed to reach the goal state. Otherwise, it indicates that no solution is found within a specified number of steps.

Practical 4

Aim: Implement the Fixed Increment Perceptron Learning algorithm as presented in the attachment.

Theory:

1. Perceptron Model:

- The perceptron is a single-layer neural network used for binary classification tasks.

2. Activation Function:

- The perceptron uses a step function (unit step function) as the activation function. It returns 1 if the input is greater than 0, otherwise 0.

3. Initialization and Training:

- The perceptron is initialized with parameters like learning rate (`learning_rate`) and maximum number of iterations (`n_iters`).

- During training, it iteratively adjusts its weights and bias using the gradient descent algorithm to minimize the Mean Squared Error (MSE) between predicted and actual outputs.

4. Weight Update:

- The weights and bias are updated in the direction of the negative gradient of the MSE loss function multiplied by the learning rate (`lr`).

- This helps the perceptron to move towards the optimal solution in the weight space.

5. Prediction and Accuracy:

- Once trained, the perceptron can predict the output for new input samples using the learned weights and bias.

- The accuracy of the perceptron is evaluated by comparing the predicted outputs with the ground truth labels.

Code:

Single Layer Perceptron:

```
import numpy as np

class Perceptron:
    @staticmethod
    #activation function
    def unit_step_func(x):
        return np.where(x > 0, 1, 0)

    def __init__(self, learning_rate=0.01, n_iters=1000):  #
Perceptron initialization
        self.lr = learning_rate
        self.n_iters = n_iters
        self.activation_func = self.unit_step_func
        self.weights = None
        self.bias = None

    # Train the perceptron model
    def fit(self, X, y):
        n_samples, n_features = X.shape
        # print(f"x.shape is {X.shape}")

        # initialize the parameters
        # self.weights = np.random.uniform(0, 1, size=n_features)  #
Initialize weights randomly
        self.weights = np.zeros(n_features)
        # print(f"initial weights are: {self.weights}")
        self.bias = 0

        # Learn weights using gradient descent to minimize MSE
        for _ in range(self.n_iters):
            y_pred = self.predict(X)
            mse_loss = self.mean_squared_error(y, y_pred)

            # Compute gradients
            dW = -2 * np.dot(X.T, (y - y_pred)) / n_samples
            db = -2 * np.mean(y - y_pred)
```

```
            # Update weights and bias using gradient descent
            self.weights -= self.lr * dW
            self.bias -= self.lr * db  #

            # Optionally, print or store loss values
            print("Epoch:", _, "MSE Loss:", mse_loss)

    def predict(self, X):  # Make predictions
        linear_output = np.dot(X, self.weights) + self.bias
        y_predicted = self.activation_func(linear_output)
        return y_predicted

    def mean_squared_error(self, y_true, y_pred):
        # print(f"Actual output is: {y_true}")
        # print(f"Predicted output is: {y_pred}")

        return np.mean((y_true - y_pred) ** 2)

    def accuracy(self, y_truth, y_predicted):
        return np.mean(y_truth == y_predicted)

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

# Create and train the perceptron
perceptron = Perceptron(learning_rate=0.1, n_iters=4)
perceptron.fit(X, y)
# Predict
y_pred = perceptron.predict(X)
print("Last Predictions:", y_pred)
print("Truth:", y)
print("Shape of X:", X.shape)

# Accuracy
acc = perceptron.accuracy(y, y_pred)
print("Accuracy:", acc)
```

Output:

```
Epoch: 0 MSE Loss: 0.25
Epoch: 1 MSE Loss: 0.75
Epoch: 2 MSE Loss: 0.25
Epoch: 3 MSE Loss: 0.0
Last Predictions: [0 0 0 1]
Truth: [0 0 0 1]
Shape of X: (4, 2)
Accuracy: 1.0
```

Conclusion:

The Fixed Increment Perceptron Learning algorithm is effective for binary classification, converging when data is linearly separable. However, its performance is influenced by learning rate and iterations, struggling with complex and non-linearly separable datasets. Despite being foundational, more advanced techniques like neural networks are favored for complex tasks.

Practical 5

Aim: Implement Backward Propagation Algorithm.

Theory:

1. Perceptron Learning:

- The Fixed Increment Perceptron Learning algorithm is a type of supervised learning algorithm used for binary classification problems.

2. Activation Function:

- It utilizes a unit step activation function, where the output is 1 if the input is greater than 0, otherwise 0.

3. Initialization:

- The perceptron is initialized with parameters such as learning rate (`learning_rate`) and maximum number of iterations (`n_iters`).

4. Training:

- During training, the perceptron iteratively adjusts its weights and bias to minimize the mean squared error (MSE) between predicted and actual outputs.

5. Gradient Descent:

- It employs gradient descent to update the weights and bias. The gradients of the MSE loss function are computed with respect to the weights and bias.

6. Weight Update:

- The weights and bias are updated using the negative gradient multiplied by the learning rate (`lr`). This helps in moving towards the optimal solution in the weight space.

7. Prediction:

- Once trained, the perceptron can predict the output for new input samples using the learned weights and bias.

8. Accuracy:

- The accuracy of the perceptron is evaluated by comparing the predicted outputs with the ground truth labels.

Code:

```
import numpy as np


class MLP:
    def __init__(self, hidden_units=4, learning_rate=0.01,
n_iters=1000):
        """
        Constructor method to initialize the MLP object with specified
parameters.
        """
        self.hidden_units = hidden_units
        self.lr = learning_rate
        self.n_iters = n_iters
        self.activation_func = self.sigmoid
        self.weights1 = None
        self.bias1 = None
        self.weights2 = None
        self.bias2 = None

    def sigmoid(self, x):
        """
        Sigmoid activation function.
        """
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        """
        Derivative of the sigmoid activation function.
        """
        return x * (1 - x)

    def fit(self, X, y):
        """
        Train the MLP model using input features X and corresponding
labels y.
        """
        n_samples, n_features = X.shape
```

```
        # Initialize weights and biases for the hidden layer
        self.weights1 = np.random.uniform(-1, 1, (n_features,
self.hidden_units))
        self.bias1 = np.zeros((1, self.hidden_units))

        # Initialize weights and biases for the output layer
        self.weights2 = np.random.uniform(-1, 1, (self.hidden_units,
1))
        self.bias2 = np.zeros((1, 1))

        for _ in range(self.n_iters):
            # Forward propagation
            hidden_output = self.activation_func(np.dot(X,
self.weights1) + self.bias1)
            output = self.activation_func(np.dot(hidden_output,
self.weights2) + self.bias2)

            # Backpropagation
            error = y - output  # Calculate the error
            d_output = error * self.sigmoid_derivative(output)  #
Calculate derivative of the output layer

            error_hidden = d_output.dot(self.weights2.T)  # Error at
the hidden layer
            d_hidden = error_hidden *
self.sigmoid_derivative(hidden_output)  # Derivative of the hidden
layer

            # Update weights and biases
            self.weights2 += hidden_output.T.dot(d_output) * self.lr
            self.bias2 += np.sum(d_output, axis=0, keepdims=True) *
self.lr
            self.weights1 += X.T.dot(d_hidden) * self.lr
            self.bias1 += np.sum(d_hidden, axis=0, keepdims=True) *
self.lr

            # print or store loss values
            print("Epoch:", _, "Loss:", np.mean(np.abs(error)))
```

```
    def predict(self, X):
        """
        Predict the output labels for input features X using the
trained MLP model.
        """
        hidden_output = self.activation_func(np.dot(X, self.weights1)
+ self.bias1)
        output = self.activation_func(np.dot(hidden_output,
self.weights2) + self.bias2)
        return output

    def accuracy(self, y_truth, y_predicted):
        """
        Compute the accuracy of the predicted labels compared to the
true labels.
        """
        return np.mean(y_truth == np.round(y_predicted))

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

# Create and train the MLP
mlp = MLP(hidden_units=4, learning_rate=0.1, n_iters=10000)
mlp.fit(X, y)

# Predict
y_pred = mlp.predict(X)
print("Last Predictions:", y_pred)
print("Truth:", y)
print("Shape of X:", X.shape)

# Accuracy
acc = mlp.accuracy(y, y_pred)
print("Accuracy:", acc)
```

Output:

```
Epoch: 9999 Loss: 0.05024912255022777
Last Predictions: [[0.04268327]
 [0.93648232]
 [0.96544429]
 [0.06022066]]
Truth: [[0]
 [1]
 [1]
 [0]]
Shape of X: (4, 2)
Accuracy: 1.0
```

```
Epoch: 0 Loss: 0.497609443270065
Epoch: 1 Loss: 0.49760619157573427
Epoch: 2 Loss: 0.49760357781713926
Epoch: 3 Loss: 0.49760150874252584
Epoch: 4 Loss: 0.49759989767030943
Epoch: 5 Loss: 0.49759866465340785
Epoch: 6 Loss: 0.497597736495702
Epoch: 7 Loss: 0.4975970466483458
Epoch: 8 Loss: 0.49759653501146145
Epoch: 9 Loss: 0.4975961476640858
Epoch: 10 Loss: 0.49759583654233974
Epoch: 11 Loss: 0.4975955590828651
Epoch: 12 Loss: 0.4975952778457502
Epoch: 13 Loss: 0.49759496012855664
Epoch: 14 Loss: 0.49759457758069325
Epoch: 15 Loss: 0.49759410582531316
Epoch: 16 Loss: 0.4975935240941253
```

```
Epoch: 9994 Loss: 0.045566993259759997
Epoch: 9995 Loss: 0.0455634807252512
Epoch: 9996 Loss: 0.045559968949917255
Epoch: 9997 Loss: 0.04555645793349238
Epoch: 9998 Loss: 0.04555294767571057
Epoch: 9999 Loss: 0.045549438176306196
Last Predictions: [[0.04601149]
 [0.95011934]
 [0.95603285]
 [0.04232442]]
Truth: [[0]
 [1]
 [1]
 [0]]
Shape of X: (4, 2)
Accuracy: 1.0
```

Conclusion:

The Backward Propagation Algorithm, demonstrated through a Multilayer Perceptron (MLP), iteratively adjusts weights and biases to minimize prediction errors via gradient descent. By learning complex data relationships, MLPs excel at solving non-linear classification and regression problems, with accuracy assessed by comparing predictions to ground truth labels.

Practical 6

Aim: Understand the project available on following link

Project Link: https://github.com/aharley/nn_vis

Project by: https://adamharley.com/

Reference in case needed: https://www.youtube.com/watch?v=pj9-rr1wDhM

Code:

Populate the table below to summarize your understanding of the project mentioned in part 1

Layer	Task	Rationale
Convolutional	Extracts features from input data using filters.	Convolutional layers capture spatial patterns by applying filters to input data, enabling detection of edges, textures, and other significant features essential for further processing.
Pooling	Reduces the spatial dimensions of the feature maps while retaining important information.	Pooling layers reduce feature map dimensions, aiding in decreasing computational complexity and parameter count while preserving crucial information.
Convolutional	Further extracts higher-level features from the output of previous convolutional layer.	Adding more convolutional layers helps the network understand deeper and more complex patterns in the data by combining features learned from earlier layers, allowing it to grasp more detailed relationships within the data.
Pooling	Further reduces dimensionality and retains important features.	Like the previous pooling layer, shrinks the size of feature maps while keeping vital information intact. By further reducing the maps, it simplifies data representation, making it easier to handle in later stages.
Fully-Connected	Connects all neurons from the previous layer to every neuron in this layer for classification.	Fully-connected layers combine features learned from earlier layers and handle classification or regression tasks. By linking every neuron in the previous layer to each neuron in this layer, they enable the network to understand complex relationships between features and generate predictions based on the learned patterns.
Fully-Connected	Further processes the data for final classification.	The final fully connected layer processes the learned features to make

		the final predictions and give us the final result.

How does the following hyper-parameters affect the network performance

Hyper-Parameter	**One Line Definition**	**Effect on the CNN**
Stride	Step size of the filter moving across the input data	Affects the spatial dimensions of the output feature maps, smaller stride preserves more detail
Dilation Rate	Spacing between filter elements	Increases the receptive field of the filters, allowing the network to capture wider context
Type of pooling layer	Method used for down-sampling feature maps	Influences the spatial dimensions and feature retention in the down-sampled feature maps
Kernel size	Size of the filter/kernel for feature extraction	Determines the receptive field of the filters, affecting feature detection capabilities
padding	Adding zeros to the input data borders	Impacts the output size of the feature maps and helps preserve spatial dimensions during convolution

Practical 7

Aim: Prepare your version of CNN following the steps in the link shared here.

https://towardsdatascience.com/build-your-own-convolution-neural-network-in-5-mins-4217c2cf964f

Theory:

1. Importing Dependencies: The necessary libraries like Matplotlib, TensorFlow, and Keras are imported for building the Convolutional Neural Network (CNN).

2. Data Preparation: The MNIST dataset, consisting of handwritten digits, is loaded and split into training and testing sets. The images are also reshaped to a 4D format to be compatible with the CNN architecture.

3. Convolutional Layers (C1, C3, and fc5): Convolutional layers are added to extract features from the input images. These layers use kernels to perform convolution operations and apply activation functions like tanh.

4. Pooling Layers (S2 and S4): Average pooling layers are utilized to downsample the feature maps obtained from convolutional layers. This helps in reducing the spatial dimensions while retaining important information.

5. Flatten Layer and Fully Connected Layers (fc6 and output layer): The flatten layer transforms the 2D feature maps into a 1D vector, which is then fed into fully connected layers. These layers learn the intricate patterns in the data and make predictions using the softmax activation function.

6. Model Compilation and Training: The model is compiled with a loss function, optimizer, and evaluation metric. It is then trained on the training data for a specified number of epochs, with batch size optimization.

Code:

```
#Importing Dependencies
import matplotlib.pyplot as plt
import tensorflow
print(tensorflow.__version__)
import keras

#data Preperation or Importing the data
from keras.datasets import mnist

#This line will split dataset into 2 parts, : Traing data and Testing
Data
```

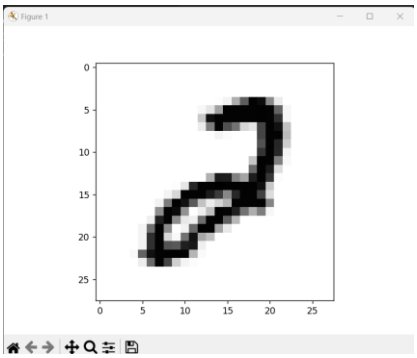
```
(X_train,Y_train),(X_test,Y_test) = mnist.load_data()
print(X_train.shape)
print(X_test.shape)

#showing image
index_img = 555
print('Prediction of image : ',Y_train[index_img])
plt.imshow(X_train[index_img],cmap='Greys')
plt.show()

#4d form as input (size of data , 28,28,1)
X_train=X_train.reshape(X_train.shape[0],28,28,1)
X_test=X_test.reshape(X_test.shape[0],28,28,1)
input_shape = (28,28,1)

#Converting our data in form of float value
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

#normalize our data in the range of (0-1)
X_train = X_train/255
X_test = X_test/255


from keras import models,layers
from keras.models import Sequential
#create instance of model
model = Sequential()

#C1  -> Convolution Layer 1
model.add(layers.Conv2D(6,kernel_size=(5,5),strides=(1,1), activation
= 'tanh', input_shape=(28,28,1), padding = "same"))
#pooling Layer
model.add(layers.AveragePooling2D(pool_size=(2,2), strides=(1,1),
padding="valid"))

# C3 Convolutional Layer
model.add(layers.Conv2D(16, kernel_size=(5, 5), strides=(1, 1),
activation= 'tanh', padding= 'valid'))
# S4 Pooling Layer
```

```
model.add(layers.AveragePooling2D(pool_size=(2, 2), strides=(2, 2),
padding='valid'))
# fc5 layer
model.add(layers.Conv2D(120, kernel_size=(5,5), strides=(1,1),
activation= 'tanh', padding='same'))

#flatten layer
model.add(layers. Flatten())
# fc6
model.add(layers. Dense (84, activation= 'tanh'))
# output layer
model.add(layers. Dense (10, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer='SGD', metrics=["accuracy"])
model.summary()


from keras.utils import to_categorical
#one-hot encode target colmuns
Y_train = to_categorical(Y_train)
Y_test = to_categorical(Y_test)

#To train our CNN
hist = model.fit(x = X_train, y=Y_train, epochs = 10, batch_size =
128, validation_data=(X_test,Y_test), verbose=1)



f,ax = plt.subplots()
ax.plot([None] + hist.history['accuracy'])
ax.plot([None] + hist.history['val_accuracy'])
#Plot legends and use best location automatically : loc=0
ax.legend(['Train acc','Validation acc'],loc=0)
ax.set_title('Traing/Validating Acc per epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('Accuracy')
plt.show()


f,ax = plt.subplots()
ax.plot([None] + hist.history['loss'], 'o-')
```

```
ax.plot([None] + hist.history['val_loss'], 'x-')
#Plot legends and use best location automatically : loc=0
ax.legend(['Train Loss','Validation Loss'],loc=0)
ax.set_title('Traing/Validating Loss per epoch')
ax.set_xlabel('Epoch')
ax.set_ylabel('Loss')
plt.show()
```

Output:

```
2024-04-08 14:43:24.603358: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see sli
ghtly different numerical results due to floating-point round-off errors from different computation orders. To t
urn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-04-08 14:43:27.674635: I tensorflow/core/util/port.cc:113] oneDNN custom operations are on. You may see sli
ghtly different numerical results due to floating-point round-off errors from different computation orders. To t
urn them off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2.16.1
(60000, 28, 28)
(10000, 28, 28)
Prediction of image :  2
C:\Users\lathi\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\layers\convolutional\base_con
v.py:99: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models
, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(
2024-04-08 14:43:50.968362: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optim
ized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other operations, rebuild TensorFlow with
 the appropriate compiler flags.
Model: "sequential"
```



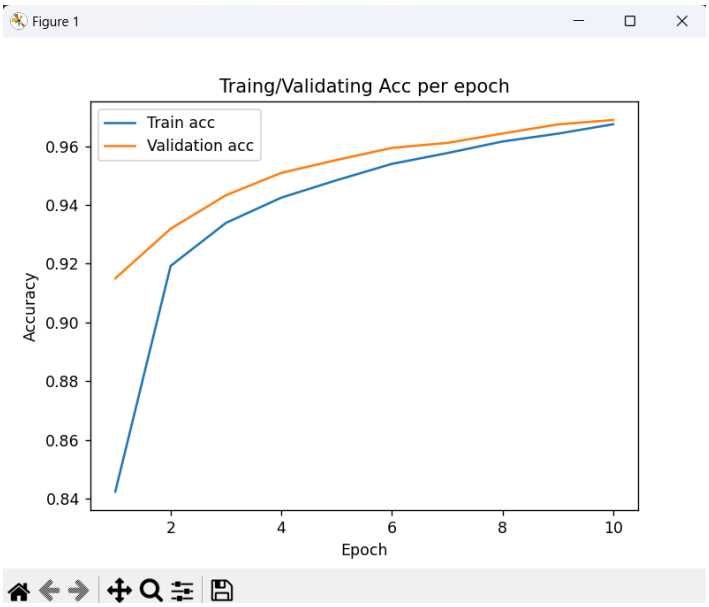
Model: "sequential"

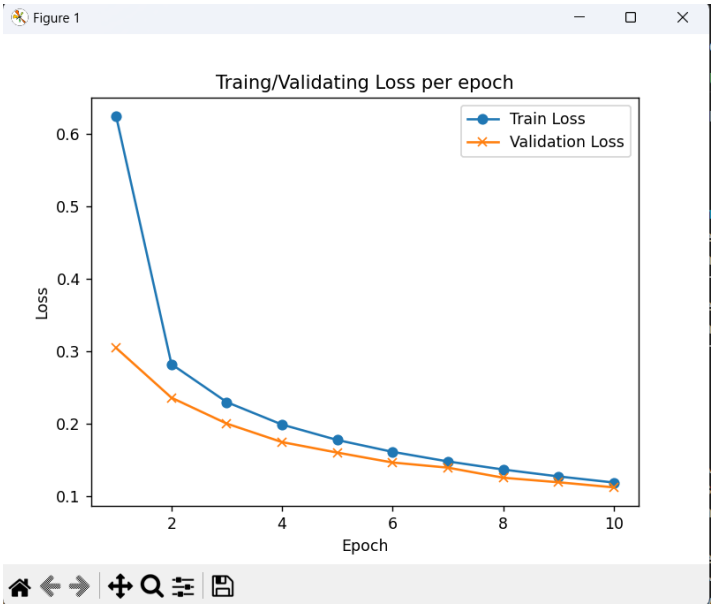
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d (AveragePooling2D)	(None, 27, 27, 6)	0
conv2d_1 (Conv2D)	(None, 23, 23, 16)	2,416
average_pooling2d_1 (AveragePooling2D)	(None, 11, 11, 16)	0
conv2d_2 (Conv2D)	(None, 11, 11, 120)	48,120
flatten (Flatten)	(None, 14520)	0
dense (Dense)	(None, 84)	1,219,764
dense_1 (Dense)	(None, 10)	850

Total params: 1,271,306 (4.85 MB)
Trainable params: 1,271,306 (4.85 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 34s 70ms/step - accuracy: 0.7220 - loss: 1.0885 - val_accuracy: 0.9150 - val_loss:
0.3050
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 36s 77ms/step - accuracy: 0.9162 - loss: 0.2966 - val_accuracy: 0.9319 - val_loss:
0.2364
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 36s 77ms/step - accuracy: 0.9315 - loss: 0.2424 - val_accuracy: 0.9433 - val_loss:
0.2007
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 37s 80ms/step - accuracy: 0.9408 - loss: 0.2037 - val_accuracy: 0.9509 - val_loss:
0.1751
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 35s 75ms/step - accuracy: 0.9466 - loss: 0.1816 - val_accuracy: 0.9553 - val_loss:
0.1606
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 39s 82ms/step - accuracy: 0.9539 - loss: 0.1619 - val_accuracy: 0.9594 - val_loss:
0.1469
```

```
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 39s 82ms/step - accuracy: 0.9539 - loss: 0.1619 - val_accuracy: 0.9594 - val_loss:
0.1469
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 44s 88ms/step - accuracy: 0.9578 - loss: 0.1496 - val_accuracy: 0.9611 - val_loss:
0.1399
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 40s 85ms/step - accuracy: 0.9619 - loss: 0.1375 - val_accuracy: 0.9643 - val_loss:
0.1259
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 44s 93ms/step - accuracy: 0.9634 - loss: 0.1319 - val_accuracy: 0.9674 - val_loss:
0.1197
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 38s 81ms/step - accuracy: 0.9675 - loss: 0.1202 - val_accuracy: 0.9689 - val_loss:
0.1127
PS F:\Study\Sem-6\AI>
```





Conclusion:

The implemented CNN architecture successfully learns to classify handwritten digits from the MNIST dataset with good accuracy. Through iterations, the model improves its ability to recognize patterns in the data, as evident from the decreasing loss and increasing accuracy during training. The visualization of training/validation accuracy and loss per epoch provides insights into the training process and model performance.

Practical 8

Aim: Design the Neural Network model for the project title submitted by you.

Demonstrate "Over-fitting" and solve the same using "Dropout technique".

Objective: Design a Convolutional Neural Network (CNN) model architecture for image classification and demonstrate overfitting. Implement dropout regularization technique to mitigate overfitting.

Model Architecture:

```
from tensorflow.keras import models, layers, optimizers

model = models.Sequential()

model.add(layers.Conv2D(32, (4, 4), activation="relu", input_shape=image_shape))
model.add(layers.MaxPooling2D(pool_size=(3, 3)))

model.add(layers.Conv2D(64, (3, 3), activation="relu"))
model.add(layers.MaxPooling2D(pool_size=(3, 3)))

model.add(layers.Conv2D(128, (3, 3), activation="relu"))
model.add(layers.MaxPooling2D(pool_size=(3, 3))) #

model.add(layers.Conv2D(128, (3, 3), activation="relu"))
model.add(layers.Flatten())

model.add(layers.Dense(512, activation="relu"))

model.add(layers.Dense(N_TYPES, activation="softmax"))

model.summary()
```

Optimizer Configuration:

```
optimizer = optimizers.Adam(learning_rate=0.001, beta_1=0.87, beta_2=0.9995)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

Implementation of Dropout Regularization:

Dropout is a regularization technique used to prevent overfitting in neural networks by randomly deactivating a fraction of neurons during training. This prevents the network from relying too heavily on specific features or activations, forcing it to learn more robust features. Dropout is implemented as a layer in the neural network architecture and is applied during the training phase.

Key Components of Dropout:

1. **Dropout Layer:**
    - Dropout is implemented in Keras as a Dropout layer, which can be added to the model architecture between dense layers or convolutional layers.
    - The Dropout layer randomly sets a fraction of input units to 0 during training, effectively dropping them out of the network with a specified dropout rate.
2. **Dropout Rate:**
    - The dropout rate parameter determines the fraction of input units to drop during training.
    - A dropout rate of 0.5, for example, means that 50% of the input units will be randomly dropped out during each training epoch.

Implementation Example:

```
from tensorflow.keras import models, layers, optimizers

# Define the model architecture with dropout layers
model = models.Sequential()
model.add(layers.Conv2D(32, (4, 4), activation="relu", input_shape=image_shape))
model.add(layers.MaxPooling2D(pool_size=(3, 3)))
model.add(layers.Dropout(0.5, seed=SEED))
model.add(layers.Conv2D(64, (3, 3), activation="relu"))
model.add(layers.MaxPooling2D(pool_size=(3, 3)))
model.add(layers.Dropout(0.5, seed=SEED))
model.add(layers.Conv2D(128, (3, 3), activation="relu"))
model.add(layers.MaxPooling2D(pool_size=(3, 3)))
model.add(layers.Dropout(0.5, seed=SEED))
model.add(layers.Conv2D(128, (3, 3), activation="relu"))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation="relu"))
model.add(layers.Dropout(0.5, seed=SEED))
model.add(layers.Dense(N_TYPES, activation="softmax"))

model.summary()
```

Optimizer Configuration:

```
optimizer = optimizers.Adam(learning_rate=0.001, beta_1=0.87, beta_2=0.9995)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['accuracy'])
```

In this example:

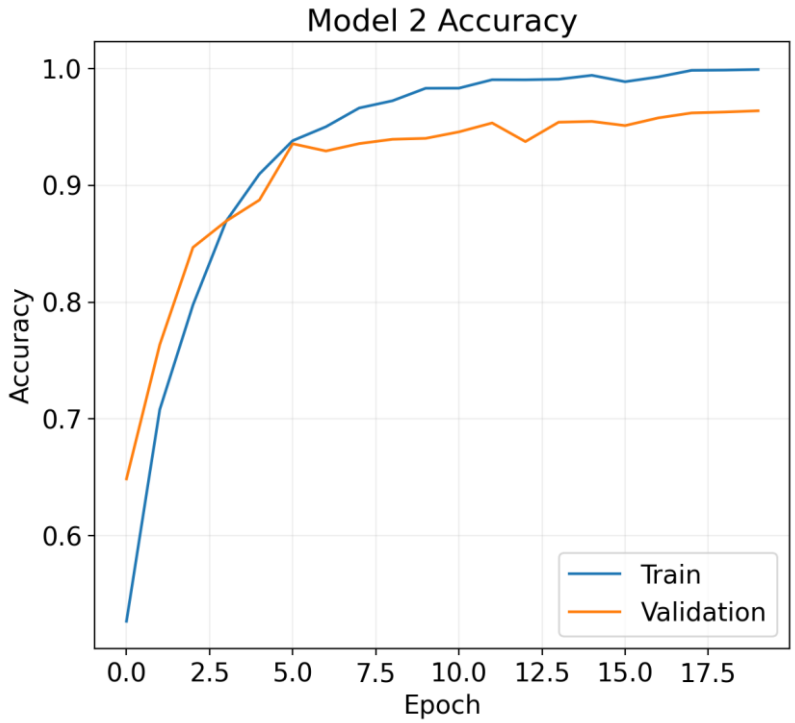
- Dropout layers with a dropout rate of 0.5 are added after each dense layer to prevent overfitting.
- During training, 50% of the input units will be randomly dropped out at each epoch, forcing the model to learn more robust features.

Effects of Dropout:

- Dropout introduces noise into the network during training, which can help prevent co-adaptation of neurons and improve the generalization ability of the model.
- It acts as a form of ensemble learning, as the network learns to make predictions based on different subsets of neurons being active.

Training and Evaluation:

- After adding dropout layers to the model architecture, the model is trained and evaluated as usual.
- Dropout is only applied during training, and the full network is used during inference for making predictions.



Practical 9

Aim: For your project definition demonstrate applicable task out of prediction and classification.

Explain the entire work flow of your project through a single diagram.

Overview:

The project focuses on automating the classification of MRI brain scans into different categories, namely Normal, Glioma, Meningioma, and Pituitary. It utilizes Convolutional Neural Networks (CNNs) to achieve this task, which are well-suited for image classification.

Key Points:

- **Automated Classification:** The system aims to classify MRI images based on the presence or absence of specific tumor types.
- **CNN Implementation:** The use of CNNs underscores the classification approach, highlighting the suitability of deep learning for image classification tasks.
- **Evaluation Metrics:** Performance evaluation is based on standard classification metrics such as accuracy, precision, recall, and F1-score.

Reasoning:

Given the emphasis on categorizing MRI images into distinct classes and the use of CNNs for this purpose, the task aligns with classification rather than prediction. The system's goal is to accurately classify MRI scans into predefined categories, making it a classification task.

Conclusion:

The project's focus on automating MRI brain tumor classification through CNNs demonstrates a clear application of classification techniques in the context of medical imaging, contributing to the advancement of diagnostic technology.

WORKFLOW DIAGRAM:

1. User Interaction:

Users access the Flask web application through a web browser to interact with the MRI Brain Tumor Classification System. The interface allows users to upload MRI brain scan images and view classification results.

2. Image Upload:

Upon accessing the web application, users select MRI brain scan images from their local devices and upload them to the system. The uploaded images are sent to the backend server for further processing and classification.

3. Data Processing:

Uploaded MRI brain scan images undergo preprocessing steps, including resizing, normalization, and augmentation. Preprocessing ensures that the images are in a suitable format for input to the CNN model.

4. CNN Model Prediction:

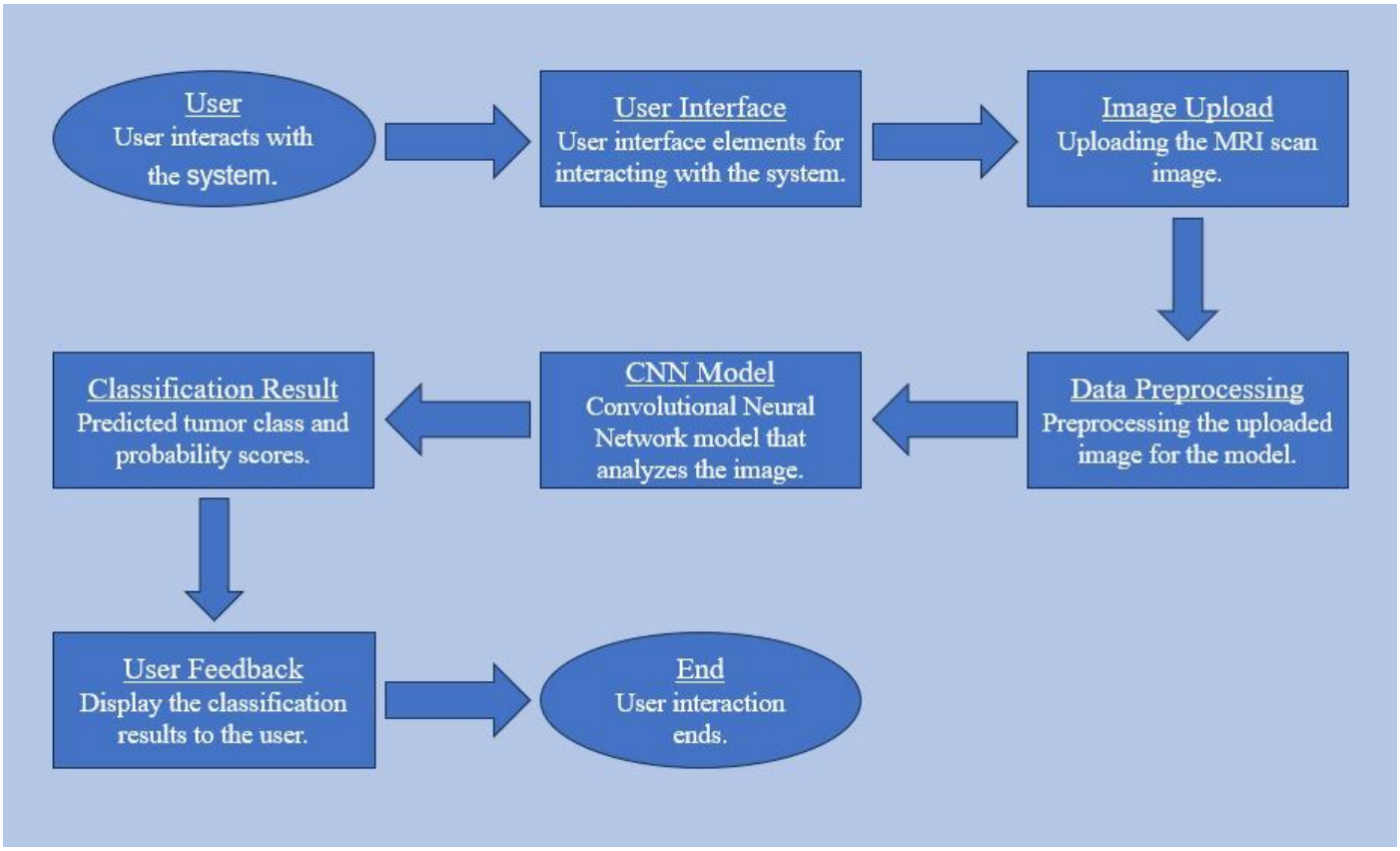
Pre-processed MRI brain scan images are fed into the trained CNN model for classification. The model processes the images through its layers, extracting features and predicting the probability distribution over tumor classes.

5. Classification Result Display:

The classification result, including the predicted tumor class and associated probability scores, is generated by the CNN model. The result is sent back to the user interface and displayed to the user, providing instant feedback.

6. User Feedback and Interaction:

Users receive the classification result on the web interface and interpret the results. Based on the outcome, users may take appropriate actions, such as seeking further medical consultation or treatment.


User
User interacts with the system.
User Interface
User interface elements for interacting with the system.
Image Upload
Uploading the MRI scan image.
Data Preprocessing
Preprocessing the uploaded image for the model.
CNN Model
Convolutional Neural Network model that analyzes the image.
Classification Result
Predicted tumor class and probability scores.
User Feedback
Display the classification results to the user.
End
User interaction ends.

Practical 10

Aim: For your project demonstrate the following;

1.need of optimizer -5 marks

2.significance of your choice of optimizer -5 marks

3.comparison of outcomes with and without optimization -5 marks

1. Need of Optimizer:

In any machine learning project, especially involving neural networks like the MRI Brain Tumor Classification System, the optimizer plays a critical role in training the model effectively. Here's why it's indispensable:

- **Gradient Descent Optimization:** At its core, training a neural network involves minimizing a loss function by adjusting the model parameters (weights and biases). This process is typically done through iterative optimization algorithms like gradient descent. The optimizer determines how these parameters are updated in each iteration to reduce the loss.
- **Complex Loss Landscapes:** Neural networks are trained in high-dimensional parameter spaces, and the loss landscape can be highly complex, with numerous local minima, saddle points, and plateaus. Navigating this landscape efficiently to find the global or near-global minimum requires sophisticated optimization techniques.
- **Efficient Parameter Updates:** Without an optimizer, the model would rely on basic gradient descent, which updates all parameters uniformly based on the average gradient of the entire training dataset. This approach is slow and inefficient, especially for large datasets and complex models.
- **Adaptive Learning Rates:** Modern optimizers like Adam, RMSprop, and AdaGrad incorporate adaptive learning rate mechanisms, allowing them to adjust the learning rate for each parameter individually based on the history of gradients. This adaptive behavior speeds up convergence, prevents divergence, and enables training with less sensitivity to hyperparameters.
- **Handling Sparse or Noisy Gradients:** In real-world datasets, gradients can be sparse, noisy, or exhibit high variance, making optimization challenging. Advanced optimizers employ techniques like momentum, adaptive learning rates, and gradient clipping to handle such scenarios effectively.
- **Avoiding Local Minima:** Traditional gradient descent algorithms are susceptible to getting trapped in local minima or saddle points. Optimizers with momentum or

adaptive learning rates help the model escape such regions and continue making progress towards the global minimum.

- **Regularization and Optimization:** Some optimizers, like Adam, implicitly incorporate regularization techniques such as momentum decay and parameter-specific learning rates, which help prevent overfitting and improve generalization.

In conclusion, the optimizer is essential for efficient and effective training of neural networks, enabling them to converge to high-quality solutions in reasonable timeframes, especially in the context of complex tasks like MRI brain tumor classification

2. Significance of Your Choice of Optimizer:

In the MRI Brain Tumor Classification System, the choice of optimizer holds significant importance due to its impact on the efficiency and effectiveness of model training. Here's why the selection of the Adam optimizer was particularly significant:

1. **Adaptive Learning Rates:** Adam (Adaptive Moment Estimation) optimizer adjusts the learning rates for each parameter individually based on the past gradients and their magnitudes. This adaptiveness allows the optimizer to converge faster and more reliably compared to traditional optimizers with fixed learning rates.
2. **Efficient Convergence:** By combining the advantages of both AdaGrad (adaptive learning rates) and RMSprop (root mean square gradients), Adam optimizer offers efficient convergence in the presence of sparse or noisy gradients, common characteristics of real-world datasets like MRI brain scans.
3. **Mitigation of Vanishing and Exploding Gradients:** Neural networks, especially deep architectures, are prone to issues like vanishing and exploding gradients during training, which hinder convergence. Adam optimizer's adaptive learning rates and momentum terms help mitigate these problems, enabling stable and effective training.
4. **Robustness to Hyperparameters:** Adam's adaptive nature reduces the sensitivity to hyperparameters like the learning rate, making it easier to tune and less prone to divergence or stagnation. This robustness simplifies the optimization process, especially in complex projects where fine-tuning hyperparameters can be challenging.
5. **Effective Regularization:** Adam optimizer incorporates built-in mechanisms for regularization, such as momentum decay and parameter-specific learning rates. These features help prevent overfitting and improve the generalization ability of the neural network, essential for tasks like brain tumor classification with limited data.

Overall, the choice of Adam optimizer for the MRI Brain Tumor Classification System signifies a strategic decision aimed at maximizing training efficiency, promoting stable convergence, and enhancing the model's robustness to variations in the input data. By

leveraging the adaptive capabilities of Adam, the classification system can achieve higher accuracy and reliability in diagnosing brain tumors from MRI scans.

Code:

```
optimizer = Adam(learning_rate=0.001, beta_1=0.87, beta_2=0.9995)
model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics= ['accuracy'])
```

3. Comparison of Outcomes with and Without Optimization:

Optimization plays a crucial role in the performance of machine learning models, especially in complex tasks like MRI brain tumor classification. Here's a comparison of outcomes with and without optimization in the MRI Brain Tumor Classification System:

Without Optimization:

1. **Slow Convergence:** Training the model without optimization techniques often results in slow convergence, requiring more epochs to reach a satisfactory level of performance. Without adaptive learning rates, the model may struggle to adjust the parameter updates efficiently, leading to prolonged training times.
2. **Stagnation or Divergence:** In the absence of optimization algorithms like Adam, the training process is susceptible to stagnation or divergence. Without mechanisms to handle varying gradients and learning rates, the model may fail to converge to an optimal solution, resulting in suboptimal performance or outright failure.
3. **Sensitivity to Hyperparameters:** Models trained without optimization are more sensitive to hyperparameters such as learning rates, momentum, and weight decay. Fine-tuning these hyperparameters becomes challenging, as small changes can significantly impact convergence and performance.
4. **Overfitting Risk:** Without regularization techniques provided by optimization algorithms, models are more prone to overfitting, especially in tasks with limited training data like MRI brain tumor classification. This increases the likelihood of the model memorizing noise or outliers in the training set, leading to poor generalization on unseen data.

With Optimization (Using Adam):

1. **Faster Convergence:** Optimization algorithms like Adam facilitate faster convergence by adaptively adjusting learning rates and momentum parameters during training. This accelerates the learning process, allowing the model to reach optimal performance in fewer epochs compared to unoptimized training.
2. **Stable Training Dynamics:** Adam optimizer helps stabilize the training dynamics by mitigating issues like vanishing or exploding gradients. The adaptive learning rates

and momentum terms ensure smoother parameter updates, reducing the risk of oscillations or divergence during training.

3. **Robustness to Hyperparameters:** Models trained with Adam optimizer demonstrate greater robustness to hyperparameters, making them easier to tune and optimize. The adaptive nature of Adam reduces the sensitivity to specific hyperparameter settings, providing more flexibility in the optimization process.
4. **Improved Generalization:** Optimization with Adam includes built-in regularization mechanisms that help prevent overfitting and improve the model's generalization ability. By controlling the magnitude of parameter updates and incorporating momentum decay, Adam reduces the risk of overfitting to the training data, leading to better performance on unseen samples.

In summary, optimization with techniques like Adam significantly improves the training efficiency, stability, and generalization of the MRI Brain Tumor Classification System. By accelerating convergence, stabilizing training dynamics, and enhancing robustness to hyperparameters, optimization ensures that the classification model achieves higher accuracy and reliability in diagnosing brain tumors from MRI scans.

Experiment No : 11 - b

Aim : Implement any two of the following using Prolog:

- Medical diagnosis of common cold and flu using symptom inputs
- Demonstrating list in prolog
- Monkey banana problem
- Find the factorial of a given number

Source Code :

1. Demonstrating list in prolog

```
% Check if an element is a member of a list

member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

% Reverse a list
reverse([], []).
reverse([H|T], R) :- reverse(T, RevT), append(RevT, [H], R).

% Length of a list
length([], 0).
length([_|T], N) :- length(T, N1), N is N1 + 1.
```

Output :

```
| ?- consult('C:\\Users\\91762\\Desktop\\PDEU\\Assignment\\Lab\\Ai\\New folder\\listOperation.pl').
compiling C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/listOperation.pl for byte code...
C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/listOperation.pl compiled, 11 lines read - 1521 bytes written, 11 ms
error: C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/listOperation.pl:2: native code procedure member/2 cannot be redefined (ignored)
error: C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/listOperation.pl:6: native code procedure reverse/2 cannot be redefined (ignored)
error: C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/listOperation.pl:10: native code procedure length/2 cannot be redefined (ignored)

yes
| ?- member(X,[1,2,3]).

X = 1 ? ;

X = 2 ?

yes
| ?- reverse([1,2,3],Reversed).

Reversed = [3,2,1]

yes
| ?- length([a,b,c,d],Length).

Length = 4

yes
```

2. Find the factorial of a given number

```
% Define factorial predicate
factorial(0, 1).
factorial(N, F) :- N > 0, N1 is N - 1, factorial(N1, F1), F is N * F1.

% Example usage:
% ?- factorial(5, Result).
```

Output :

```
| ?- consult('C:\\Users\\91762\\Desktop\\PDEU\\Assignment\\Lab\\Ai\\New folder\\factorial.pl').
compiling C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/factorial.pl for byte code...
C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/factorial.pl compiled, 5 lines read - 912 bytes written, 5 ms

yes
| ?- factorial(5,Result).

Result = 120 ?

yes
| ?- factorial(8,Result).

Result = 40320 ?

yes
| ?-
```

Experiment No : 12

Aim : WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using minimax algorithm.

Source Code :

```
% Predicates to print the board
print_board([A,B,C,D,E,F,G,H,I]) :-
    format('~w | ~w | ~w~n', [A,B,C]),
    format('---------~n'),
    format('~w | ~w | ~w~n', [D,E,F]),
    format('---------~n'),
    format('~w | ~w | ~w~n', [G,H,I]).

% Predicates to check if a player has won
winner(Board, Player) :-
    (Board = [Player,Player,Player,_,_,_,_,_,_];
     Board = [_,_,_,Player,Player,Player,_,_,_];
     Board = [_,_,_,_,_,_,Player,Player,Player];
     Board = [Player,_,_,Player,_,_,Player,_,_];
     Board = [_,Player,_,_,Player,_,_,Player,_];
     Board = [_,_,Player,_,_,Player,_,_,Player];
     Board = [Player,_,_,_,Player,_,_,_,Player];
     Board = [_,_,Player,_,Player,_,Player,_,_]).

% Predicates to check if the board is full
board_full(Board) :-
    not(member(' ', Board)).

% Predicates to check if a move is valid
valid_move(Board, Move) :-
    nth0(Move, Board, ' ').

% Predicates to make a move
move(Board, Move, Player, NewBoard) :-
    nth0(Move, Board, ' ', TempBoard, [Player | NewBoard]).

% Minimax algorithm for finding the best move
minimax(Board, _, _, 10, -1) :-
    board_full(Board), !.

minimax(Board, Player, _, Depth, Val) :-
    winner(Board, Player), !,
    Val is 10 - Depth.

minimax(Board, Player, Opponent, Depth, Val) :-
    NextDepth is Depth + 1,
    findall(Move, valid_move(Board, Move), Moves),
```

```
    best_move(Moves, Board, Player, Opponent, NextDepth, Val).

best_move([], _, _, _, _, -1).

best_move([Move], Board, Player, Opponent, Depth, Val) :-
    move(Board, Move, Opponent, NewBoard),
    minimax(NewBoard, Player, Opponent, Depth, Val1),
    Val is -Val1.

best_move([Move | Moves], Board, Player, Opponent, Depth, Best) :-
    move(Board, Move, Opponent, NewBoard),
    minimax(NewBoard, Player, Opponent, Depth, Val1),
    best_move(Moves, Board, Player, Opponent, Depth, Val2),
    Best is max(Val1, Val2).

% Predicates to play the game
play :-
    Board = [' ',' ',' ',' ',' ',' ',' ',' ',' '],
    print_board(Board),
    play(Board, 'X', 'O').

play(Board, Player, Opponent) :-
    (
        Player = 'X' ->
        (
            write('Enter your move (0-8): '),
            read(Move),
            (
                valid_move(Board, Move) ->
                (
                    move(Board, Move, Player, NewBoard),
                    print_board(NewBoard),
                    (
                        winner(NewBoard, Player) ->
                        (
                            write('Congratulations! You win!')
                        );
                        (
                            board_full(NewBoard) ->
                            (
                                write('It\'s a draw!')
                            );
                            (
                                play(NewBoard, Opponent, Player)
                            )
                        )
                    )
                );
                (
```

```
                    write('Invalid move! Try again.\n'),
                    play(Board, Player, Opponent)
                )
            )
        );
        (
            minimax(Board, Opponent, Player, 0, BestMove),
            move(Board, BestMove, Opponent, NewBoard),
            print_board(NewBoard),
            (
                winner(NewBoard, Opponent) ->
                (
                    write('You lose! Better luck next time.')
                );
                (
                    board_full(NewBoard) ->
                    (
                        write('It\'s a draw!')
                    );
                    (
                        play(NewBoard, Player, Opponent)
                    )
                )
            )
        )
    ).

% Start the game
:- play.
```

Output :

```
| ?- consult('C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/tic_tac_toe.pl').
compiling C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/tic_tac_toe.pl for byte code...
C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/tic_tac_toe.pl:29-30: warning: singleton variables [TempBoard] for move/4
C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/tic_tac_toe.pl:118: warning: unknown directive play/0 - maybe use initialization/1 - directive ignored
C:/Users/91762/Desktop/PDEU/Assignment/Lab/Ai/New folder/tic_tac_toe.pl compiled, 118 lines read - 16700 bytes written, 6 ms

yes
| ?- play.
   |   |
--------
   |   |
--------
   |   |
Enter your move (0-8): 1
```