SCHOOL OF TECHNOLOGY

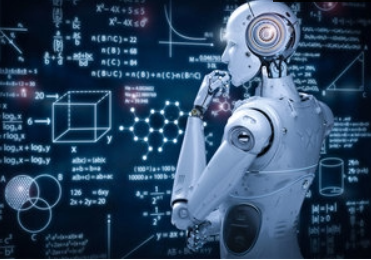
PANDIT DEENDAYAL ENERGY UNIVERSITY GANDHINAGAR, GUJARAT, INDIA

Computer Science & Engineering

LAB File (2022-23)

ARTIFICIAL INTELLIGENCE LAB

(20CP313P)



**Student Name: Suyamoon Pathak**

**Roll No.: 20BCP298 Course with Semester: 6**

**Division: 4**

**Group: 8**

**Instructor: Dr. Shilpa Pandey**

**SCHOOL OF TECHNOLOGY**

**PANDIT DEENDAYAL ENERGY UNIVERSITY GANDHINAGAR, GUJARAT, INDIA**

**CERTIFICATE**

This is to certify that Mr. /Miss Suyamoon Pathak

Roll no 20BCP298 of **6th Semester** degree course in **Computer**

**Science and Engineering** has satisfactorily prepared and presented his / her Term Work in **Artificial Intelligence (20CP313P)** within four walls of the laboratory of this Institute during January 2023 to May 2023.

**Date of Submission: 28 April 2023**

**Dr. Shakti Mishra Dr. Shilpa Pandey**

**Head of Department**  **Assistant Professor**

**List of Practical**

|  |  |  |  |
| --- | --- | --- | --- |
| **Exp. No.** | **Experiment Title** | **Date** | **Signature** |
| 1 | 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. |  |  |
| 2 | You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Make a rule-based solution to this water jug problem. |  |  |
| 3 | You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Use DFS to solve water jug problem. |  |  |
| 4 | You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Use BFS to solve water jug problem. |  |  |
| 5 | WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using rule-based solution. |  |  |
| 6 | WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using minimax algorithm or any smart solution. |  |  |
| 7 | Solve 8 puzzle problems using BFS or DFS where initial state, goal state and name of the method will be given by the users. |  |  |
| 8 | WAP to implement A\* algorithm 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. |  |  |
| 9 | Solve 8 puzzle problem using A\* algorithm where initial state and Goal state will be given by the users. |  |  |
| 10 | Study of Prolog programming and its function. (a) insertion (b) deletion (c) concatenation (d) sub list ( e) membership |  |  |
| 11 | Write a program to solve the Monkey Banana problem. |  |  |
| 12 | WAP to implement AND, NAND, OR, NOR, XOR logic Gate using perceptron neural network. |  |  |
| 13 | Design Machine Learning model for the house price prediction. To train models, use data available on the below link. <https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data> |  |  |
| 14 | Design a Machine Learning model for the Heart Attack prediction.  Download the Dataset from MS team along with the description |  |  |
| 15 | Design a Convolutional Neural Network from Scratch for MNIST fashion dataset. Apply dropout technique to deal with the overfitting. Dataset can be downloaded from the below link. <https://www.kaggle.com/datasets/zalando-research/fashionmnist> |  |  |

Experiment-1

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.

from collections import deque

# Define the graph as an adjacency list

graph = {

    'S': ['A', 'B'],

    'A': ['C', 'D'],

    'B': ['E', 'F'],

    'C': [],

    'D': ['G'],

    'E': [],

    'F': [],

    'G': []

}

def bfs(graph, source, goal):

    # Create a queue for BFS and enqueue the source node

    queue = deque([source])

    # Create a dictionary to store the path from the source to each node

    paths = {source: [source]}

    while queue:

        # Dequeue a node from the queue

        node = queue.popleft()

        # If the node is the goal node, return the path from the source to the goal

        if node == goal:

            return paths[node]

        # Enqueue all adjacent nodes that have not been visited yet

        for neighbor in graph[node]:

            if neighbor not in paths:

                paths[neighbor] = paths[node] + [neighbor]

                queue.append(neighbor)

    # If the goal node is not found, return None

    return None

def dfs(graph, source, goal):

    # Create a stack for DFS and push the source node

    stack = [source]

    # Create a dictionary to store the path from the source to each node

    paths = {source: [source]}

    while stack:

        # Pop a node from the stack

        node = stack.pop()

        # If the node is the goal node, return the path from the source to the goal

        if node == goal:

            return paths[node]

        # Push all adjacent nodes that have not been visited yet

        for neighbor in reversed(graph[node]):

            if neighbor not in paths:

                paths[neighbor] = paths[node] + [neighbor]

                stack.append(neighbor)

    # If the goal node is not found, return None

    return None

# Get the source and goal nodes from the user

source = input('Enter the source node: ')

goal = input('Enter the goal node: ')

# Traverse the graph using BFS and print the result

path = bfs(graph, source, goal)

if path is not None:

    print('BFS path:', ' -> '.join(path))

else:

    print('No path found using BFS.')

# Traverse the graph using DFS and print the result

path = dfs(graph, source, goal)

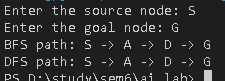
if path is not None:

    print('DFS path:', ' -> '.join(path))

else:

    print('No path found using DFS.')

Output



Experiment-2

You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Make a rule-based solution to this water jug problem.

import time

start=time.time()

def getNextStates(state):

    nextStates = []

    if(state[0] < m):

        nextStates.append((m,state[1]))

    if(state[1] < n):

        nextStates.append((state[0],n))

    if(state[0] != 0):

        nextStates.append((0, state[1]))

    if(state[1] != 0):

        nextStates.append((state[0], 0))

    if((state[0] + state[1] >= m) and state[1] > 0):

        nextStates.append((m, state[1] - (m- state[0])))

    if((state[0]+state[1] >= n) and state[0] > 0):

        nextStates.append((state[0] - (n - state[1]), n))

    if((state[0]+state[1] < m) and state[1] > 0):

        nextStates.append((state[0] + state[1], 0))

    if((state[0]+state[1] < n) and state[0] > 0):

        nextStates.append((0, state[0] + state[1]))

    return nextStates

m = int(input('enter m-'))

n = int(input('enter n-'))

x = 0

y = 0

goalstate = int(input('enter goalstate-'))

visited = set()

queue = [(x,y)]

path = []

flag = 0

while(len(queue) > 0):

    node = queue.pop(0)

    if(node in visited):

        continue

    visited.add(node)

    path.append(node)

    if(node[0] == goalstate or node[1] == goalstate):

        print("Success ", path)

        flag = 1

        break

    nextStates = getNextStates(node)

    for state in nextStates:

        if(state not in visited):

            queue.append(state)

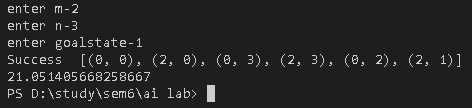
if(flag == 0):

    print("Solution not found")

end=time.time()

print(end-start)

Output



Experiment-3

You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Use DFS to solve water jug problem.

 import time

start = time.time()

def gcd(a,b):

  if a==0:

    return b

  else:

    return gcd(b%a,a)

m=int(input("Enter the capacity of first jug-"))

n=int(input("Enter the capacity of second jug-"))

d=int(input("Enter the goal number-"))

if d % gcd(m,n) != 0:

    print("Solution Not possible")

else:

    x=0

    y=0

    a=m

    b=n

    if(b>a):

        a, b = b, a

    while(x+y!=d):

        print(f"{x} {y}")

        if(x==0):

            x=a

            continue

        if(y!=b):

            temp = min(x,b-y)

            x-= temp

            y+= temp

            continue

        if(y==b):

            y=0

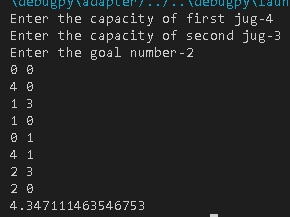
            continue

    print(f"{x} {y}")

    end = time.time()

    print(end-start)

Output



Experiment-4

You are given two jugs with m liters and a n liter 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 liters of water where d is less than n. Use BFS to solve water jug problem.

from collections import deque

def BFS(a, b, target):

    m = {}

    isSolvable = False

    path = []

    q = deque()

    q.append((0, 0))

    while (len(q) > 0):

        u = q.popleft()

        if ((u[0], u[1]) in m):

            continue

        if ((u[0] > a or u[1] > b or u[0] < 0 or u[1] < 0)):

            continue

        path.append([u[0], u[1]])

        m[(u[0], u[1])] = 1

        if (u[0] == target or u[1] == target):

            isSolvable = True

            if (u[0] == target):

                if (u[1] != 0):

                    path.append([u[0], 0])

            else:

                if (u[0] != 0):

                    path.append([0, u[1]])

            sz = len(path)

            for i in range(sz):

                print("(", path[i][0], ",", path[i][1], ")")

            break

        q.append([u[0], b])

        q.append([a, u[1]])

        for ap in range(max(a, b) + 1):

            c = u[0] + ap

            d = u[1] - ap

            if (c == a or (d == 0 and d >= 0)):

                q.append([c, d])

            c = u[0] - ap

            d = u[1] + ap

            if ((c == 0 and c >= 0) or d == b):

                q.append([c, d])

        q.append([a, 0])

        q.append([0, b])

    if (not isSolvable):

        print("No solution")

if \_\_name\_\_ == '\_\_main\_\_':

    Jug1=(int(input('enter capacity of jug1-')))

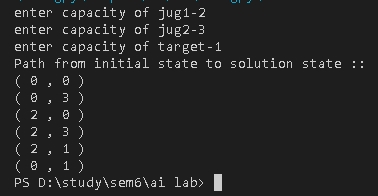
    Jug2=(int(input('enter capacity of jug2-')))

    target=(int(input('enter capacity of target-')))

    print("Path from initial state " "to solution state ::")

    BFS(Jug1, Jug2, target)

Output



Experiment-5

WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using rule-based solution.

# Import the random module for the AI's moves

import random

# Initialize the board as an empty dictionary

board = {'1': ' ', '2': ' ', '3': ' ',

         '4': ' ', '5': ' ', '6': ' ',

         '7': ' ', '8': ' ', '9': ' '}

# Define the function to print the board

def print\_board():

    print(board['1'] + '|' + board['2'] + '|' + board['3'])

    print('-+-+-')

    print(board['4'] + '|' + board['5'] + '|' + board['6'])

    print('-+-+-')

    print(board['7'] + '|' + board['8'] + '|' + board['9'])

# Define the function to check if someone has won

def check\_win(player):

    # Check rows

    if board['1'] == board['2'] == board['3'] == player:

        return True

    elif board['4'] == board['5'] == board['6'] == player:

        return True

    elif board['7'] == board['8'] == board['9'] == player:

        return True

    # Check columns

    elif board['1'] == board['4'] == board['7'] == player:

        return True

    elif board['2'] == board['5'] == board['8'] == player:

        return True

    elif board['3'] == board['6'] == board['9'] == player:

        return True

    # Check diagonals

    elif board['1'] == board['5'] == board['9'] == player:

        return True

    elif board['3'] == board['5'] == board['7'] == player:

        return True

    else:

        return False

# Define the function for the AI's move

def ai\_move():

    # Check if the AI can win in the next move

    for i in range(1, 10):

        if board[str(i)] == ' ':

            board[str(i)] = 'O'

            if check\_win('O'):

                return

            else:

                board[str(i)] = ' '

    # Check if the player can win in the next move and block them

    for i in range(1, 10):

        if board[str(i)] == ' ':

            board[str(i)] = 'X'

            if check\_win('X'):

                board[str(i)] = 'O'

                return

            else:

                board[str(i)] = ' '

    # Choose a random empty square

    while True:

        i = random.randint(1, 9)

        if board[str(i)] == ' ':

            board[str(i)] = 'O'

            return

# Define the main function to play the game

def play\_game():

    print("Welcome to Tic Tac Toe!")

    print("Player 1 is X and Player 2 is O.")

    print("The board is numbered from 1 to 9 as shown below:")

    print\_board()

    current\_player = 'X'

    moves = 0

    while moves < 9:

        if current\_player == 'X':

            print("It's " + current\_player + "'s turn. Please enter a number (1-9) to make your move:")

            move = input()

            if move not in board.keys() or board[move] != ' ':

                print("Invalid move. Please try again.")

                continue

            board[move] = current\_player

        else:

            print("It's " + current\_player + "'s turn.")

            ai\_move()

        print\_board()

        if check\_win(current\_player):

            print(current\_player + " wins!")

            return

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

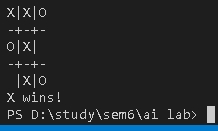
        moves += 1

    print("It's a tie!")

# Start the game

play\_game()

Output



Experiment-6

WAP to design Tic Tac Toe games from O (Opponent) and X (Player) by using minimax algorithm or any smart solution.

def display\_board(board):

    print('   |   |')

    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])

    print('   |   |')

    print('-----------')

    print('   |   |')

    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])

    print('   |   |')

    print('-----------')

    print('   |   |')

    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])

    print('   |   |')

def player\_input():

    marker = ''

    while not (marker == 'X' or marker == 'O'):

        marker = input('Player 1: Do you want to be X or O? ').upper()

    if marker == 'X':

        return ('X', 'O')

    else:

        return ('O', 'X')

def place\_marker(board, marker, position):

    board[position] = marker

def win\_check(board,mark):

    return ((board[7] == mark and board[8] == mark and board[9] == mark) or

    (board[4] == mark and board[5] == mark and board[6] == mark) or

    (board[1] == mark and board[2] == mark and board[3] == mark) or

    (board[7] == mark and board[4] == mark and board[1] == mark) or

    (board[8] == mark and board[5] == mark and board[2] == mark) or

    (board[9] == mark and board[6] == mark and board[3] == mark) or

    (board[7] == mark and board[5] == mark and board[3] == mark) or

    (board[9] == mark and board[5] == mark and board[1] == mark))

import random

def choose\_first():

    if random.randint(0, 1) == 0:

        return 'Player 2'

    else:

        return 'Player 1'

def getBoardCopy(board):

     dupeBoard = []

     for i in board:

         dupeBoard.append(i)

     return dupeBoard

def space\_check(board, position):

    return board[position] == ' '

def full\_board\_check(board):

    for i in range(1,10):

        if space\_check(board, i):

            return False

    return True

def chooseRandomMoveFromList(board, movesList):

     possibleMoves = []

     for i in movesList:

         if space\_check(board, i):

             possibleMoves.append(i)

     if len(possibleMoves) != 0:

         return random.choice(possibleMoves)

     else:

         return None

def player\_choice(board):

    position = 0

    while position not in [1,2,3,4,5,6,7,8,9] or not space\_check(board, position):

        position = int(input('Choose your next position: (1-9) '))

    return position

def computer\_choice(board):

    position = 0

    while position not in [1,2,3,4,5,6,7,8,9] or not space\_check(board, position):

        position = random.randint(1,9)

    return position

def replay():

    return input('Do you want to play again? Enter Yes or No: ').lower().startswith('y')

def getComputerMove(board, computerLetter):

     if computerLetter == 'X':

         playerLetter = 'O'

     else:

         playerLetter = 'X'

     for i in range(1, 10):

         copy = getBoardCopy(board)

         if space\_check(copy, i):

             place\_marker(copy, computerLetter, i)

             if win\_check(copy, computerLetter):

                 return i

     for i in range(1, 10):

         copy = getBoardCopy(board)

         if space\_check(copy, i):

             place\_marker(copy, playerLetter, i)

             if win\_check(copy, playerLetter):

                 return i

     move = chooseRandomMoveFromList(board, [1, 3, 7, 9])

     if move != None:

         return move

     if space\_check(board, 5):

         return 5

     return chooseRandomMoveFromList(board, [2, 4, 6, 8])

print('Welcome to Tic Tac Toe!')

while True:

    theBoard = [' '] \* 10

    player1\_marker, comp\_marker = player\_input()

    turn = choose\_first()

    print(turn + ' will go first.')

    play\_game = input('Are you ready to play? Enter Yes or No.')

    if play\_game.lower()[0] == 'y':

        game\_on = True

    else:

        game\_on = False

    while game\_on:

        if turn == 'Player 1':

            display\_board(theBoard)

            position = player\_choice(theBoard)

            place\_marker(theBoard, player1\_marker, position)

            if win\_check(theBoard, player1\_marker):

                display\_board(theBoard)

                print('Congratulations! You have won the game!')

                game\_on = False

            else:

                if full\_board\_check(theBoard):

                    display\_board(theBoard)

                    print('The game is a draw!')

                    break

                else:

                    turn = 'Player 2'

        else:

            display\_board(theBoard)

            position = getComputerMove(theBoard,comp\_marker)

            place\_marker(theBoard, comp\_marker, position)

            if win\_check(theBoard, comp\_marker):

                display\_board(theBoard)

                print('Player 2 has won!')

                game\_on = False

            else:

                if full\_board\_check(theBoard):

                    display\_board(theBoard)

                    print('The game is a draw!')

                    break

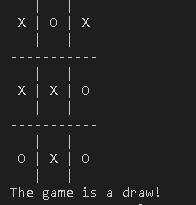
                else:

                    turn = 'Player 1'

    if not replay():

        break

Output



Experiment-7

Solve 8 puzzle problems using BFS or DFS where initial state, goal state and name of the method will be given by the users.

from collections import deque

class Node:

    def \_\_init\_\_(self, state, parent=None, action=None):

        self.state = state

        self.parent = parent

        self.action = action

    def expand(self):

        """Returns a list of all valid actions and resulting states."""

        children = []

        state = self.state

        i = state.index(0)

        if i > 2:

            # move up

            s = state[:]

            s[i], s[i-3] = s[i-3], s[i]

            children.append(Node(s, self, 'Up'))

        if i < 6:

            # move down

            s = state[:]

            s[i], s[i+3] = s[i+3], s[i]

            children.append(Node(s, self, 'Down'))

        if i % 3 > 0:

            # move left

            s = state[:]

            s[i], s[i-1] = s[i-1], s[i]

            children.append(Node(s, self, 'Left'))

        if i % 3 < 2:

            # move right

            s = state[:]

            s[i], s[i+1] = s[i+1], s[i]

            children.append(Node(s, self, 'Right'))

        return children

def bfs(initial\_state, goal\_state):

    frontier = deque([Node(initial\_state)])

    explored = set()

    steps = 0

    while frontier:

        node = frontier.popleft()

        explored.add(tuple(node.state))

        if node.state == goal\_state:

            # trace back the solution path

            path = []

            while node.parent:

                path.append(node.action)

                node = node.parent

            path.reverse()

            return path, steps, len(explored)

        for child in node.expand():

            if tuple(child.state) not in explored:

                frontier.append(child)

                steps += 1

    return None, steps, len(explored)

def dfs(initial\_state, goal\_state):

    frontier = [Node(initial\_state)]

    explored = set()

    steps = 0

    while frontier:

        node = frontier.pop()

        explored.add(tuple(node.state))

        if node.state == goal\_state:

            # trace back the solution path

            path = []

            while node.parent:

                path.append(node.action)

                node = node.parent

            path.reverse()

            return path, steps, len(explored)

        for child in node.expand()[::-1]:

            if tuple(child.state) not in explored:

                frontier.append(child)

                steps += 1

    return None, steps, len(explored)

# take input from user

initial\_state = list(map(int, input("Enter initial state: ").split()))

goal\_state = list(map(int, input("Enter goal state: ").split()))

method = input("Enter method (BFS or DFS): ")

if method.lower() == 'bfs':

    path, steps, nodes\_visited = bfs(initial\_state, goal\_state)

elif method.lower() == 'dfs':

    path, steps, nodes\_visited = dfs(initial\_state, goal\_state)

else:

    print("Invalid method.")

    exit()

if path is None:

    print("No solution found.")

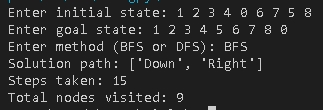
else:

    print("Solution path:", path)

    print("Steps taken:", steps)

    print("Total nodes visited:", nodes\_visited)

Output



Experiment-8

WAP to implement A\* algorithm 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.

from queue import PriorityQueue

# Define the graph as an adjacency list

graph = {

    'S': {'A': 1, 'B': 3},

    'A': {'C': 5, 'D': 3},

    'B': {'E': 6, 'F': 2},

    'C': {'G': 6},

    'D': {'G': 2},

    'E': {},

    'F': {},

    'G': {}

}

def astar(graph, source, goal):

    # Create a priority queue for the open list and enqueue the source node

    open\_list = PriorityQueue()

    open\_list.put((0, source))

    # Create a dictionary to store the path from the source to each node

    paths = {source: [source]}

    # Create a dictionary to store the cost of the path from the source to each node

    g\_scores = {source: 0}

    while not open\_list.empty():

        # Dequeue the node with the lowest f-score from the open list

        f\_score, node = open\_list.get()

        # If the node is the goal node, return the path from the source to the goal

        if node == goal:

            return paths[node]

        # Expand the node and enqueue all adjacent nodes that have not been visited yet

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

            # Calculate the tentative g-score for the neighbor

            tentative\_g\_score = g\_scores[node] + cost

            # If the neighbor has not been visited or the tentative g-score is lower than the previous g-score

            if neighbor not in g\_scores or tentative\_g\_score < g\_scores[neighbor]:

                # Update the path and g-score for the neighbor

                paths[neighbor] = paths[node] + [neighbor]

                g\_scores[neighbor] = tentative\_g\_score

                # Calculate the f-score for the neighbor and enqueue it in the open list

                h\_score = 0  # Use a heuristic function here if desired

                f\_score = tentative\_g\_score + h\_score

                open\_list.put((f\_score, neighbor))

    # If the goal node is not found, return None

    return None

# Get the source and goal nodes from the user

source = input('Enter the source node: ')

goal = input('Enter the goal node: ')

# Traverse the graph using A\* and print the result

path = astar(graph, source, goal)

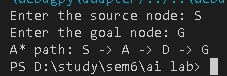
if path is not None:

    print('A\* path:', ' -> '.join(path))

else:

    print('No path found using A\*.')

Output



Experiment-9

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

from copy import deepcopy

import numpy as np

import time

# takes the input of current states and evaluvates the best path to goal state

def bestsolution(state):

    bestsol = np.array([], int).reshape(-1, 9)

    count = len(state) - 1

    while count != -1:

        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)

        count = (state[count]['parent'])

    return bestsol.reshape(-1, 3, 3)

# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.

def all(checkarray):

    set=[]

    for it in set:

        for checkarray in it:

            return 1

        else:

            return 0

# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state

def manhattan(puzzle, goal):

    a = abs(puzzle // 3 - goal // 3)

    b = abs(puzzle % 3 - goal % 3)

    mhcost = a + b

    return sum(mhcost[1:])

# will calcuates the number of misplaced tiles in the current state as compared to the goal state

def misplaced\_tiles(puzzle,goal):

    mscost = np.sum(puzzle != goal) - 1

    return mscost if mscost > 0 else 0

#3[on\_true] if [expression] else [on\_false]

# will indentify the coordinates of each of goal or initial state values

def coordinates(puzzle):

    pos = np.array(range(9))

    for p, q in enumerate(puzzle):

        pos[q] = p

    return pos

# start of 8 puzzle evaluvation, using Manhattan heuristics

def evaluvate(puzzle, goal):

    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],

                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

     # initializing the parent, gn and hn, where hn is manhattan distance function call

    costg = coordinates(goal)

    parent = -1

    gn = 0

    hn = manhattan(coordinates(puzzle), costg)

    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.

    dtpriority = [('position', int),('fn', int)]

    priority = np.array( [(0, hn)], dtpriority)

    while 1:

        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

        position, fn = priority[0]

        priority = np.delete(priority, 0, 0)

        # sort priority queue using merge sort,the first element is picked for exploring remove from queue what we are exploring

        puzzle, parent, gn, hn = state[position]

        puzzle = np.array(puzzle)

        # Identify the blank square in input

        blank = int(np.where(puzzle == 0)[0])

        gn = gn + 1

        c = 1

        start\_time = time.time()

        for s in steps:

            c = c + 1

            if blank not in s['position']:

                # generate new state as copy of current

                openstates = deepcopy(puzzle)

                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

                # The all function is called, if the node has been previously explored or not

                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

                    end\_time = time.time()

                    if (( end\_time - start\_time ) > 2):

                        print(" The 8 puzzle is unsolvable ! \n")

                        exit

                    # calls the manhattan function to calcuate the cost

                    hn = manhattan(coordinates(openstates), costg)

                     # generate and add new state in the list

                    q = np.array([(openstates, position, gn, hn)], dtstate)

                    state = np.append(state, q, 0)

                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state

                    fn = gn + hn

                    q = np.array([(len(state) - 1, fn)], dtpriority)

                    priority = np.append(priority, q, 0)

                      # Checking if the node in openstates are matching the goal state.

                    if np.array\_equal(openstates, goal):

                        print(' The 8 puzzle is solvable ! \n')

                        return state, len(priority)

    return state, len(priority)

# start of 8 puzzle evaluvation, using Misplaced tiles heuristics

def evaluvate\_misplaced(puzzle, goal):

    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],

                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)

    # initializing the parent, gn and hn, where hn is misplaced\_tiles  function call

    parent = -1

    gn = 0

    hn = misplaced\_tiles(coordinates(puzzle), costg)

    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   # We make use of priority queues with position as keys and fn as value.

    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)

    while 1:

        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])

        position, fn = priority[0]

        # sort priority queue using merge sort,the first element is picked for exploring.

        priority = np.delete(priority, 0, 0)

        puzzle, parent, gn, hn = state[position]

        puzzle = np.array(puzzle)

         # Identify the blank square in input

        blank = int(np.where(puzzle == 0)[0])

        # Increase cost g(n) by 1

        gn = gn + 1

        c = 1

        start\_time = time.time()

        for s in steps:

            c = c + 1

            if blank not in s['position']:

                 # generate new state as copy of current

                openstates = deepcopy(puzzle)

                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']], openstates[blank]

                # The check function is called, if the node has been previously explored or not.

                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():

                    end\_time = time.time()

                    if (( end\_time - start\_time ) > 2):

                        print(" The 8 puzzle is unsolvable \n")

                        break

                    # calls the Misplaced\_tiles function to calcuate the cost

                    hn = misplaced\_tiles(coordinates(openstates), costg)

                    # generate and add new state in the list

                    q = np.array([(openstates, position, gn, hn)], dtstate)

                    state = np.append(state, q, 0)

                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the goal state

                    fn = gn + hn

                    q = np.array([(len(state) - 1, fn)], dtpriority)

                    priority = np.append(priority, q, 0)

                    # Checking if the node in openstates are matching the goal state.

                    if np.array\_equal(openstates, goal):

                        print(' The 8 puzzle is solvable \n')

                        return state, len(priority)

    return state, len(priority)

# ----------  Program start -----------------

 # User input for initial state

puzzle = []

print(" Input vals from 0-8 for start state ")

for i in range(0,9):

    x = int(input("enter vals :"))

    puzzle.append(x)

 # User input of goal state

goal = []

print(" Input vals from 0-8 for goal state ")

for i in range(0,9):

    x = int(input("Enter vals :"))

    goal.append(x)

n = int(input("1. Manhattan distance \n2. Misplaced tiles\n"))

if(n ==1 ):

    state, visited = evaluvate(puzzle, goal)

    bestpath = bestsolution(state)

    print(str(bestpath).replace('[', ' ').replace(']', ''))

    totalmoves = len(bestpath) - 1

    print('Steps to reach goal:',totalmoves)

    visit = len(state) - visited

    print('Total nodes visited: ',visit, "\n")

    print('Total generated:', len(state))

if(n == 2):

    state, visited = evaluvate\_misplaced(puzzle, goal)

    bestpath = bestsolution(state)

    print(str(bestpath).replace('[', ' ').replace(']', ''))

    totalmoves = len(bestpath) - 1

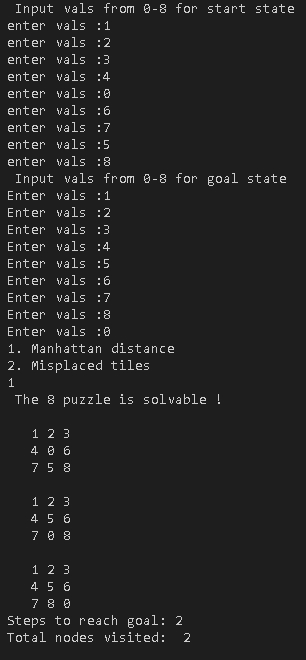
    print('Steps to reach goal:',totalmoves)

    visit = len(state) - visited

    print('Total nodes visited: ',visit, "\n")

    print('Total generated:', len(state))

Output



Experiment-10

Study of Prolog programming and its function. (a) insertion (b) deletion (c) concatenation (d) sub list ( e) membership

Insertion

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

list\_insert(X,L,R) :- list\_delete(X,R,L).

Output



Deletion

list\_delete(X, [X], []).

list\_delete(X,[X|L1], L1).

list\_delete(X, [Y|L2], [Y|L1]) :- list\_delete(X,L2,L1).

Output



Concatenation

list\_concat([],L,L).

list\_concat([X1|L1],L2,[X1|L3]) :- list\_concat(L1,L2,L3).

Output



Sublist

sublist([], \_).

sublist([H|T], L) :-

append(\_, [H|Sub], L),

sublist(T, Sub).

Output



Membership

list\_member(X,[X|\_]).

list\_member(X,[\_|TAIL]) :- list\_member(X,TAIL).

Output



Experiment-11

Write a program to solve the Monkey Banana problem.

movee(state(middle,onbox,middle,hasnot),

grasp,

state(middle,onbox,middle,has)).

movee(state(P,onfloor,P,H),

climb,

state(P,onbox,P,H)).

movee(state(P1,onfloor,P1,H),

drag(P1,P2),

state(P2,onfloor,P2,H)).

movee(state(P1,onfloor,B,H),

walk(P1,P2),

state(P2,onfloor,B,H)).

canget(state(\_,\_,\_,has)).

canget(State1) :- movee(State1,\_,State2),canget(State2).

Ouput



Experiment-12

WAP to implement AND, NAND, OR, NOR, XOR logic Gate using perceptron neural network.

AND

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# AND Logic Function

# w1 = 1, w2 = 1, b = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    b = -1.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

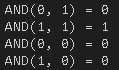
print("AND({}, {}) = {}".format(0, 1, AND\_logicFunction(test1)))

print("AND({}, {}) = {}".format(1, 1, AND\_logicFunction(test2)))

print("AND({}, {}) = {}".format(0, 0, AND\_logicFunction(test3)))

print("AND({}, {}) = {}".format(1, 0, AND\_logicFunction(test4)))

Output



NAND

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 0

    else:

        return 1

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

    wNOT = -1

    bNOT = 0.5

    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# w1 = 1, w2 = 1, bAND = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    bAND = -1.5

    return perceptronModel(x, w, bAND)

# NAND Logic Function

# with AND and NOT

# function calls in sequence

def NAND\_logicFunction(x):

    output\_AND = AND\_logicFunction(x)

    output\_NOT = NOT\_logicFunction(output\_AND)

    return output\_NOT

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

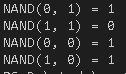
print("NAND({}, {}) = {}".format(0, 1, NAND\_logicFunction(test1)))

print("NAND({}, {}) = {}".format(1, 1, NAND\_logicFunction(test2)))

print("NAND({}, {}) = {}".format(0, 0, NAND\_logicFunction(test3)))

print("NAND({}, {}) = {}".format(1, 0, NAND\_logicFunction(test4)))

Output



OR

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# OR Logic Function

# w1 = 1, w2 = 1, b = -0.5

def OR\_logicFunction(x):

    w = np.array([1, 1])

    b = -0.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

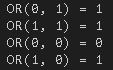
print("OR({}, {}) = {}".format(0, 1, OR\_logicFunction(test1)))

print("OR({}, {}) = {}".format(1, 1, OR\_logicFunction(test2)))

print("OR({}, {}) = {}".format(0, 0, OR\_logicFunction(test3)))

print("OR({}, {}) = {}".format(1, 0, OR\_logicFunction(test4)))

Output



NOR

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 0

    else:

        return 1

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOR Logic Function

# w1 = 1, w2 = 1, b = -0.5

def OR\_logicFunction(x):

    w = np.array([1, 1])

    b = -0.5

    return perceptronModel(x, w, b)

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

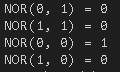
print("NOR({}, {}) = {}".format(0, 1, OR\_logicFunction(test1)))

print("NOR({}, {}) = {}".format(1, 1, OR\_logicFunction(test2)))

print("NOR({}, {}) = {}".format(0, 0, OR\_logicFunction(test3)))

print("NOR({}, {}) = {}".format(1, 0, OR\_logicFunction(test4)))

Output



XOR

# importing Python library

import numpy as np

# define Unit Step Function

def unitStep(v):

    if v >= 0:

        return 1

    else:

        return 0

# design Perceptron Model

def perceptronModel(x, w, b):

    v = np.dot(w, x) + b

    y = unitStep(v)

    return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

    wNOT = -1

    bNOT = 0.5

    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# here w1 = wAND1 = 1,

# w2 = wAND2 = 1, bAND = -1.5

def AND\_logicFunction(x):

    w = np.array([1, 1])

    bAND = -1.5

    return perceptronModel(x, w, bAND)

# OR Logic Function

# w1 = 1, w2 = 1, bOR = -0.5

def OR\_logicFunction(x):

    w = np.array([1, 1])

    bOR = -0.5

    return perceptronModel(x, w, bOR)

# XOR Logic Function

# with AND, OR and NOT

# function calls in sequence

def XOR\_logicFunction(x):

    y1 = AND\_logicFunction(x)

    y2 = OR\_logicFunction(x)

    y3 = NOT\_logicFunction(y1)

    final\_x = np.array([y2, y3])

    finalOutput = AND\_logicFunction(final\_x)

    return finalOutput

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

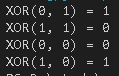
print("XOR({}, {}) = {}".format(0, 1, XOR\_logicFunction(test1)))

print("XOR({}, {}) = {}".format(1, 1, XOR\_logicFunction(test2)))

print("XOR({}, {}) = {}".format(0, 0, XOR\_logicFunction(test3)))

print("XOR({}, {}) = {}".format(1, 0, XOR\_logicFunction(test4)))

Output



Experiment-13

Design Machine Learning model for the house price prediction. To train models, use data available on the below link. <https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data>

import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow import keras

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

from sklearn.metrics import mean\_squared\_error

# load data

train\_df = pd.read\_csv('train.csv')

test\_df = pd.read\_csv('test.csv')

# preprocess data

# drop columns with high percentage of missing values

train\_df = train\_df.drop(['Alley', 'PoolQC', 'Fence', 'MiscFeature'], axis=1)

test\_df = test\_df.drop(['Alley', 'PoolQC', 'Fence', 'MiscFeature'], axis=1)

# fill missing values

train\_df.fillna(train\_df.mean(), inplace=True)

test\_df.fillna(test\_df.mean(), inplace=True)

# convert categorical features to numerical

train\_df = pd.get\_dummies(train\_df)

test\_df = pd.get\_dummies(test\_df)

# align train and test data

train\_labels = train\_df['SalePrice']

train\_df, test\_df = train\_df.align(test\_df, join='inner', axis=1)

# split data into train and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(train\_df, train\_labels, test\_size=0.2, random\_state=42)

# build ANN model

model = keras.Sequential([

    keras.layers.Dense(128, activation='relu', input\_shape=(X\_train.shape[1],)),

    keras.layers.Dense(64, activation='relu'),

    keras.layers.Dense(1)

])

# compile the model

model.compile(loss='mean\_squared\_error',

              optimizer='adam',

              metrics=['accuracy', 'mean\_squared\_error'])

# train the model

history = model.fit(X\_train, y\_train,

                    epochs=10,

                    batch\_size=32,

                    verbose=1,

                    validation\_data=(X\_val, y\_val))

# evaluate model performance on training set

train\_pred = model.predict(X\_train)

train\_mse = mean\_squared\_error(y\_train, train\_pred)

train\_acc = model.evaluate(X\_train, y\_train, verbose=0)[1]

# evaluate model performance on validation set

val\_pred = model.predict(X\_val)

val\_mse = mean\_squared\_error(y\_val, val\_pred)

val\_acc = model.evaluate(X\_val, y\_val, verbose=0)[1]

# print results

print('Training accuracy:', train\_acc)

print('Testing accuracy:', val\_acc)

print('mean squared error:', val\_mse)

# predict house prices for test set

test\_preds = model.predict(test\_df)

output



Experiment-14

Design a Machine Learning model for the Heart Attack prediction.

Download the Dataset from MS team along with the description

import pandas as pd

import numpy as np

import tensorflow as tf

from tensorflow import keras

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

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score, f1\_score

# load data

df = pd.read\_csv('Heart.csv')

# preprocess data

# drop rows with missing values

df = df.dropna()

# split data into features and target

X = df.drop(['target'], axis=1)

y = df['target']

# normalize data

X = (X - X.mean()) / X.std()

# split data into train and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# build ANN model

model = keras.Sequential([

    keras.layers.Dense(128, activation='relu', input\_shape=(X\_train.shape[1],)),

    keras.layers.Dense(64, activation='relu'),

    keras.layers.Dense(1, activation='sigmoid')

])

# compile the model

model.compile(loss='binary\_crossentropy',

              optimizer='adam',

              metrics=['accuracy', keras.metrics.Precision(), keras.metrics.Recall()])

# train the model

history = model.fit(X\_train, y\_train,

                    epochs=10,

                    batch\_size=32,

                    verbose=1,

                    validation\_data=(X\_val, y\_val))

# evaluate model performance on validation set

y\_pred = model.predict(X\_val)

y\_pred = np.round(y\_pred).flatten()

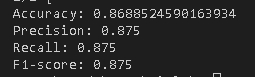
print('Accuracy:', accuracy\_score(y\_val, y\_pred))

print('Precision:', precision\_score(y\_val, y\_pred))

print('Recall:', recall\_score(y\_val, y\_pred))

print('F1-score:', f1\_score(y\_val, y\_pred))

output



Experiment-15

Design a Convolutional Neural Network from Scratch for MNIST fashion dataset. Apply dropout technique to deal with the overfitting. Dataset can be downloaded from the below link. <https://www.kaggle.com/datasets/zalando-research/fashionmnist>

Import libraries

In [ ]:

**import** numpy **as** np

**import** pandas **as** pd

**import** tensorflow **as** tf

**import** matplotlib.pyplot **as** plt

**import** seaborn **as** sns

**from** tensorflow.keras **import** layers, models

**from** tensorflow.keras.datasets **import** fashion\_mnist

Load MNIST fashion dataset

In [ ]:

(x\_train, y\_train), (x\_test, y\_test) **=** fashion\_mnist**.**load\_data()

Preprocessing

In [ ]:

x\_train **=** x\_train**.**astype('float32') **/** 255.0

x\_test **=** x\_test**.**astype('float32') **/** 255.0

y\_train **=** tf**.**keras**.**utils**.**to\_categorical(y\_train, 10)

y\_test **=** tf**.**keras**.**utils**.**to\_categorical(y\_test, 10)

Defining CNN architecture

In [ ]:

model **=** models**.**Sequential()

model**.**add(layers**.**Conv2D(32, (3, 3), activation**=**'relu', input\_shape**=**(28, 28, 1)))

model**.**add(layers**.**MaxPooling2D((2, 2)))

model**.**add(layers**.**Conv2D(64, (3, 3), activation**=**'relu'))

model**.**add(layers**.**MaxPooling2D((2, 2)))

model**.**add(layers**.**Conv2D(64, (3, 3), activation**=**'relu'))

model**.**add(layers**.**Flatten())

model**.**add(layers**.**Dense(64, activation**=**'relu'))

model**.**add(layers**.**Dropout(0.5))

model**.**add(layers**.**Dense(10, activation**=**'softmax'))

Model Compilation

In [ ]:

model**.**compile(optimizer**=**'adam',

loss**=**'categorical\_crossentropy',

metrics**=**['accuracy'])

Model Training

In [ ]:

history **=** model**.**fit(x\_train**.**reshape(**-**1, 28, 28, 1), y\_train, epochs**=**10,

validation\_data**=**(x\_test**.**reshape(**-**1, 28, 28, 1), y\_test))

Epoch 1/10

1875/1875 [==============================] - 38s 19ms/step - loss: 0.6746 - accuracy: 0.7549 - val\_loss: 0.4137 - val\_accuracy: 0.8468

Epoch 2/10

1875/1875 [==============================] - 38s 20ms/step - loss: 0.4320 - accuracy: 0.8455 - val\_loss: 0.3541 - val\_accuracy: 0.8718

Epoch 3/10

1875/1875 [==============================] - 41s 22ms/step - loss: 0.3666 - accuracy: 0.8714 - val\_loss: 0.3089 - val\_accuracy: 0.8863

Epoch 4/10

1875/1875 [==============================] - 44s 24ms/step - loss: 0.3283 - accuracy: 0.8851 - val\_loss: 0.2990 - val\_accuracy: 0.8929

Epoch 5/10

1875/1875 [==============================] - 45s 24ms/step - loss: 0.3038 - accuracy: 0.8938 - val\_loss: 0.2951 - val\_accuracy: 0.8965

Epoch 6/10

1875/1875 [==============================] - 45s 24ms/step - loss: 0.2816 - accuracy: 0.9007 - val\_loss: 0.2922 - val\_accuracy: 0.8947

Epoch 7/10

1875/1875 [==============================] - 48s 26ms/step - loss: 0.2625 - accuracy: 0.9064 - val\_loss: 0.2778 - val\_accuracy: 0.9018

Epoch 8/10

1875/1875 [==============================] - 49s 26ms/step - loss: 0.2503 - accuracy: 0.9117 - val\_loss: 0.2745 - val\_accuracy: 0.9057

Epoch 9/10

1875/1875 [==============================] - 52s 28ms/step - loss: 0.2366 - accuracy: 0.9145 - val\_loss: 0.2650 - val\_accuracy: 0.9080

Epoch 10/10

1875/1875 [==============================] - 52s 28ms/step - loss: 0.2250 - accuracy: 0.9185 - val\_loss: 0.2742 - val\_accuracy: 0.9044

Model Evaluation

In [ ]:

test\_loss, test\_acc **=** model**.**evaluate(x\_test**.**reshape(**-**1, 28, 28, 1), y\_test)

print('Test accuracy:', test\_acc)

313/313 [==============================] - 2s 7ms/step - loss: 0.2742 - accuracy: 0.9044

Test accuracy: 0.9043999910354614

Graph

**import** matplotlib.pyplot **as** plt

*# Creating a figure object and creating twin axes*

fig, ax1 **=** plt**.**subplots()

ax2 **=** ax1**.**twinx()

*# Plotting the training loss and accuracy on respective axes*

ax1**.**plot(history**.**history['loss'], color**=**'blue', label**=**'Training Loss')

ax2**.**plot(history**.**history['accuracy'], color**=**'green', label**=**'Training Accuracy')

*# Adding title and labels*

ax1**.**set\_title('Training Metrics')

ax1**.**set\_xlabel('Epochs')

ax1**.**set\_ylabel('Loss')

ax2**.**set\_ylabel('Accuracy')

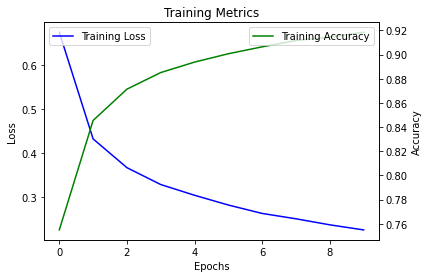
*# Adding legend for both axes*

ax1**.**legend(loc**=**'upper left')

ax2**.**legend(loc**=**'upper right')

*# Displaying the plot*

plt**.**show()



model summary

In [ ]:

model**.**summary()

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 26, 26, 32) 320

max\_pooling2d (MaxPooling2D (None, 13, 13, 32) 0

)

conv2d\_1 (Conv2D) (None, 11, 11, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d (Conv2D) (None, 26, 26, 32) 320

max\_pooling2d (MaxPooling2D (None, 13, 13, 32) 0

)

conv2d\_1 (Conv2D) (None, 11, 11, 64) 18496

max\_pooling2d\_1 (MaxPooling (None, 5, 5, 64) 0

2D)

conv2d\_2 (Conv2D) (None, 3, 3, 64) 36928

flatten (Flatten) (None, 576) 0

dense (Dense) (None, 64) 36928

dropout (Dropout) (None, 64) 0

dense\_1 (Dense) (None, 10) 650

=================================================================

Total params: 93,322

Trainable params: 93,322

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_