Experiment 6

AIM : Write a program in python to implement neural networks.

THEORY : Neural networks are a foundational theory in artificial intelligence that mimic the structure and function of the human brain to process data and learn patterns. They consist of layers of interconnected nodes (neurons) that transform input data through weighted connections and activation functions. As data passes through these layers, the network learns to make predictions or classifications by adjusting weights via training algorithms like backpropagation. Neural networks power many modern AI applications, including image recognition, natural language processing, and autonomous systems.

CODE :

```
import numpy as np


# Activation function (Sigmoid)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))


# Derivative of Sigmoid
def sigmoid_derivative(x):
    return x * (1 - x)


# Neural Network class
class NeuralNetwork:
    def _init_(self, input_nodes, hidden_nodes, output_nodes):
        self.input_nodes = input_nodes
        self.hidden_nodes = hidden_nodes
        self.output_nodes = output_nodes


        # Initialize weights and biases
        self.weights_input_hidden = np.random.rand(self.input_nodes,
self.hidden_nodes)
        self.weights_hidden_output = np.random.rand(self.hidden_nodes,
self.output_nodes)
```

```
        self.bias_hidden = np.random.rand(self.hidden_nodes)
        self.bias_output = np.random.rand(self.output_nodes)

    def feedforward(self, X):
        # Forward propagation
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) +
self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

        self.final_input = np.dot(self.hidden_layer_output, self.weights_hidden_output)
+ self.bias_output
        self.final_output = sigmoid(self.final_input)
        return self.final_output

    def train(self, X, y, learning_rate, epochs):
        for _ in range(epochs):
            # Forward pass
            self.feedforward(X)

            # Backpropagation
            output_error = y - self.final_output
            output_delta = output_error * sigmoid_derivative(self.final_output)

            hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
            hidden_delta = hidden_error * sigmoid_derivative(self.hidden_layer_output)

            # Update weights and biases
            self.weights_hidden_output += np.dot(self.hidden_layer_output.T,
output_delta) * learning_rate
            self.bias_output += np.sum(output_delta, axis=0) * learning_rate
```

```
        self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0) * learning_rate

# Example usage
if _name_ == "_main_":
    # Sample dataset (X: inputs, y: outputs)
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    y = np.array([[0], [1], [1], [0]]) # XOR problem

    nn = NeuralNetwork(input_nodes=2, hidden_nodes=4, output_nodes=1)
    nn.train(X, y, learning_rate=0.5, epochs=10000)

    # Test the network
    print("Predictions:")
    print(nn.feedforward(X))
```

OUTPUT :

```
Predictions:
[[0.01502373]
 [0.98154121]
 [0.98974501]
 [0.01589742]]
```

EXPERIMENT 7

AIM : Compare the different search algorithms.

Theory : Short Note on BFS, DFS, UCS, and A* Search:

- **Breadth-First Search (BFS):**
  BFS explores nodes level by level, starting from the source. It uses a queue and is guaranteed to find the shortest path in an unweighted graph. It is complete and optimal for such graphs.
- **Depth-First Search (DFS):**
  DFS explores as far as possible along each branch before backtracking. It uses a stack (or recursion) and is memory efficient, but it may not find the shortest path and can get stuck in deep or infinite paths without precautions.
- **Uniform Cost Search (UCS):**
  UCS expands the node with the lowest total cost from the start. It is optimal and complete for graphs with positive edge weights, making it suitable for finding the least-cost path.
- **A* Search:**
  A* enhances UCS by using a heuristic to estimate the cost to the goal. It selects paths with the lowest combined actual cost and estimated future cost, making it efficient and optimal with an admissible heuristic.

CODE :

```
from queue import PriorityQueue
def bfs(graph, start, goal):

    queue = [(start, [start])]
    while queue:
        node, path = queue.pop(0)
        if node == goal:
        return path
        for neighbor in graph[node]:
            if neighbor not in path:
                queue.append((neighbor, path + [neighbor]))
    return None

def dfs(graph, start, goal, path=[]):
    path = path + [start]
    if start == goal:
        return path
    for neighbor in graph[start]:
        if neighbor not in path:
            new_path = dfs(graph, neighbor, goal, path)
            if new_path:
                return new_path
    return None
```

```
def ucs(graph, start, goal):
   queue = PriorityQueue()
   queue.put((0, start, [start]))
   while not queue.empty():
      cost, node, path = queue.get()
      if node == goal:
         return path
      for neighbor, weight in graph[node]:
         if neighbor not in path:
            queue.put((cost + weight, neighbor, path + [neighbor]))
   return None

def heuristic(node, goal):
   return abs(ord(node) - ord(goal)) # Example heuristic function

def a_star(graph, start, goal):
queue = PriorityQueue()
queue.put((0, start, [start]))
while not queue.empty():
      cost, node, path = queue.get()
      if node == goal:
         return path
      for neighbor, weight in graph[node]:
         if neighbor not in path:
            total_cost = cost + weight + heuristic(neighbor, goal)
            queue.put((total_cost, neighbor, path + [neighbor]))
   return None

# Graph representation
graph_unweighted = {
   'A': ['B', 'C'],
   'B': ['A', 'D', 'E'],
   'C': ['A', 'F'],
   'D': ['B'],
   'E': ['B', 'F'],
   'F': ['C', 'E']
}
graph_weighted = {

   'A': [('B', 1), ('C', 4)],
   'B': [('A', 1), ('D', 2), ('E', 5)],
   'C': [('A', 4), ('F', 3)],
   'D': [('B', 2)],
   'E': [('B', 5), ('F', 1)],
   'F': [('C', 3), ('E', 1)]
}   #   Running   algorithms   print("BFS:",
bfs(graph_unweighted,  'A',  'F'))  print("DFS:",
dfs(graph_unweighted, 'A', 'F'))
```

```
print("UCS:", ucs(graph_weighted, 'A', 'F'))
print("A*:", a_star(graph_weighted, 'A', 'F'))
```

OUTPUT:

```
...  Comparing Search Algorithms:

     Search Algorithm Comparison:
     ------------------------------------------------------------
     Algorithm                 Path                 Execution Time (s)
     ------------------------------------------------------------
     Breadth-First Search (BFS) ['A', 'C', 'F']      0.000000
     Depth-First Search (DFS)  ['A', 'B', 'E', 'F'] 0.000000
     Uniform Cost Search (UCS) ['A', 'C', 'F']      0.000000
     A* Search                 ['A', 'C', 'F']      0.000000
```

EXPERIMENT 8

AIM: WAP to implement factorial, Fibonacci of a given number in PROLOG.

THEORY :

Factorial In Prolog : The factorial of a number N is the product of all positive integers from 1 to N, and is denoted as N!. In Prolog, this is implemented using recursion with a base case factorial(0, 1) which returns 1, since 0! is defined as 1. The recursive rule states that the factorial of N is N multiplied by the factorial of N-1, allowing Prolog to compute the result by breaking down the problem into smaller subproblems until the base case is reached.

Fibonacci in Prolog : The Fibonacci sequence is a series of numbers where each term is the sum of the two preceding ones, starting from 0 and 1. In Prolog, this is defined using two base cases: fibonacci(0, 0) and fibonacci(1, 1), which represent the first two terms of the sequence. For values greater than 1, the recursive rule computes the Fibonacci number by summing the results of fibonacci(N-1) and fibonacci(N-2), building up the sequence until the desired position is reached.

CODE :

Factorial in Prolog :

```
% Base case: factorial of 0 is 1

factorial(0, 1).



% Recursive case: factorial of N is N * factorial of (N-1)

factorial(N, F) :-

    N > 0,

    N1 is N - 1,

    factorial(N1, F1),

    F is N * F1.
```

OUTPUT :

```
?- factorial(5, F).
F = 120.
```

Fibonacci in Prolog :

% Base case: Fibonacci of 0 is 0, Fibonacci of 1 is 1

fibonacci(0, 0). fibonacci(1, 1).

% Recursive case: Fibonacci of N is Fibonacci(N-1) + Fibonacci(N-2)

```
fibonacci(N, F) :-
    N > 1, N1 is N - 1,
    N2 is N - 2,
    fibonacci(N1, F1),
    fibonacci(N2, F2),
    F is F1 + F2.
```

OUTPUT :

```
?- fibonacci(5, F).
F = 5.
```

EXPERIMENT 9

AIM : WAP to solve 4-queen problem in PROLOG.

THEORY : The 4-Queens problem is a specific case of the classic N-Queens problem in which the goal is to place 4 queens on a 4×4 chessboard such that no two queens threaten each other. A queen in chess can move any number of squares vertically, horizontally, or diagonally, so the solution must ensure that no two queens share the same row, column, or diagonal. In Prolog, this problem is solved using a **permutation-based backtracking approach**, where each permutation of the list [1, 2, 3, 4] represents a possible placement of queens in different columns, one in each row. The program checks each permutation to determine if it is **safe**, i.e., no two queens are on the same diagonal, by comparing the difference in row and column indices. This approach leverages Prolog's powerful pattern matching and recursion to generate and test all valid configurations efficiently, ultimately producing only those placements where no queens attack each other.

CODE :

```
% Main predicate to solve N-Queens
queens(N, Solution) :-
   range(1, N, Range),
   permutation(Range, Solution),
   safe(Solution).

% Generate list from From to To
range(From, To, [From|Rest]) :-
   From =< To,
   Next is From + 1,
   range(Next, To, Rest).
range(From, To, []) :-
   From > To.

% Check if the board is safe (no queens attacking each other diagonally)
safe([]).
safe([Q|Others]) :-
   safe(Q, Others, 1),
   safe(Others).
```

```
% Check that no two queens attack each other diagonally
safe(_, [], _).
safe(Q, [Q1|Others], D) :-
    Q =\= Q1 + D,
    Q =\= Q1 - D,
    D1 is D + 1,
    safe(Q, Others, D1).
```

QUERY :

```
?- queens(4, Solution).
```

OUTPUT :

```
Solution = [2, 4, 1, 3] ;
Solution = [3, 1, 4, 2] ;
false.
```

EXPERIMENT 10

AIM : WAP to implement fuzzy logic.

THEORY : Fuzzy Logic is a computational approach that deals with reasoning under uncertainty . Unlike classical logic which uses binary true (1) or false (0), fuzzy logic allows values between 0 and 1 , representing degrees of truth.It is based on fuzzy sets and is used to handle vague or imprecise information, much like human reasoning.

CODE :

```
def cold_membership(temp):

    if temp <= 10:

        return 1.0

    elif 10 < temp < 20:

        return (20 - temp) / 10.0

    else:

        return 0.0



def warm_membership(temp):

    if 15 <= temp <= 25:

        return (temp - 15) / 10.0

    elif 25 < temp <= 35:

        return (35 - temp) / 10.0

    else:

        return 0.0



def hot_membership(temp):

    if temp <= 30:
```

```
        return 0.0
    elif 30 < temp < 40:
        return (temp - 30) / 10.0
    else:
        return 1.0

def calculate_fan_speed(temp):
    cold = cold_membership(temp)
    warm = warm_membership(temp)
    hot = hot_membership(temp)

    # Weighted average (centroid method)
    if cold + warm + hot == 0:
        return 0 # Avoid division by zero

    fan_speed = (cold * 0 + warm * 50 + hot * 100) / (cold + warm + hot)

    print("\nFuzzy Membership Values:")
    print(f"Cold : {cold:.2f}")
    print(f"Warm : {warm:.2f}")
    print(f"Hot : {hot:.2f}")
    print(f"\nCalculated Fan Speed: {fan_speed:.2f}%")
    return fan_speed
```

```
# Main program

if _name_ == "_main_":

    temp = float(input("Enter temperature (in Celsius): "))

    calculate_fan_speed(temp)
```

OUTPUT :

```
Enter temperature (in Celsius): 25

Fuzzy Membership Values:
Cold : 0.00
Warm : 1.00
Hot  : 0.00

Calculated Fan Speed: 50.00%
```

EXPERIMENT 11

AIM: WAP to implement genetic algorithm.

THEORY : A **Genetic Algorithm (GA)** is a heuristic search algorithm inspired by the principles of natural selection and genetics. It is used to find approximate solutions to optimization and search problems. GA belongs to a class of algorithms known as **evolutionary algorithms**, which simulate the process of natural evolution.

CODE :

```
import random


# Parameters
POPULATION_SIZE = 6
CHROMOSOME_LENGTH = 5 # To represent 0–31 in binary
MUTATION_RATE = 0.1
GENERATIONS = 10


# Fitness function: f(x) = x^2
def fitness(chromosome):
x = int(chromosome, 2)
return x ** 2


# Generate a random chromosome
def random_chromosome():
    return ''.join(random.choice(['0', '1']) for _ in range(CHROMOSOME_LENGTH))


# Selection: Tournament selection
def selection(population):
    selected = random.sample(population, 2)
    return max(selected, key=fitness)


# Crossover: Single point crossover
```

```
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:] child2 =
    parent2[:point] + parent1[point:] return child1, child2


# Mutation: Flip bits with some probability
def mutate(chromosome):
    return ''.join(
        bit if random.random() > MUTATION_RATE else ('1' if bit == '0' else '0')
        for bit in chromosome
    )

# Genetic Algorithm
def genetic_algorithm():
    # Step 1: Initialize population
    population = [random_chromosome() for _ in range(POPULATION_SIZE)]

    for generation in range(GENERATIONS):
    print(f"\nGeneration {generation + 1}:")
    population = sorted(population, key=fitness, reverse=True)

        # Display best in current generation
        best = population[0]
        print(f"Best: {best} -> x={int(best, 2)} fitness={fitness(best)}")

        # Step 2: Create new generation
        new_population = population[:2] # Elitism: carry forward best 2

        while len(new_population) < POPULATION_SIZE:
```

```
            parent1 = selection(population) parent2 =
            selection(population) child1, child2 =
            crossover(parent1, parent2)
            new_population.append(mutate(child1)) if
            len(new_population) < POPULATION_SIZE:
                new_population.append(mutate(child2))

        population = new_population

    # Final result
    best = max(population, key=fitness)
    print(f"\nBest solution after {GENERATIONS} generations:")
    print(f"Chromosome: {best} -> x={int(best, 2)}, fitness={fitness(best)}")

    # Run the GA
    genetic_algorithm()
```

OUTPUT :

```
...  Generation 0: Best Solution = 8.0126, Fitness = 7.9120
     Generation 10: Best Solution = 7.9787, Fitness = 7.9167
     Generation 20: Best Solution = 7.9787, Fitness = 7.9167
     Generation 30: Best Solution = 7.9787, Fitness = 7.9167
     Generation 40: Best Solution = 7.9787, Fitness = 7.9167
     Generation 50: Best Solution = 7.9787, Fitness = 7.9167
     Generation 60: Best Solution = 7.9787, Fitness = 7.9167
     Generation 70: Best Solution = 7.9787, Fitness = 7.9167
     Generation 80: Best Solution = 7.9787, Fitness = 7.9167
     Generation 90: Best Solution = 7.9787, Fitness = 7.9167

     Final Results:
     Best Solution: 7.9787
     Best Fitness: 7.9167
```