Jaihind Comprehensive Educational Institute`s

JAIHIND COLLEGE OF ENGINEERING

Tal- Junnar, Dist- Pune, Pin Code -410511

DTE Code: EN6609, SPPU Code: CEGP015730

Computer Laboratory III [417534]

LABORATORY MANUAL

DEPARMTENT OF ARTIFICIAL INTELLIGENCE &

DATA SCIENCE ENGINEERING

Institute Vision

• To enrich the role of nation building by imparting the qualitative technical education.

Institute Mission

• Impart technical knowledge through prescribed curriculum of university.

• Inculcate ethical and moral values in students for environmental and sustainable development.

• Equip the aspirants through co-curricular and extra- curricular activities to excel in career.

Department Vision

• To provide current technical education and train competitive engineering professionals in Artificial Intelligence and Data Science with a commitment to fulfilling industry requirements.

Department Mission

• To foster students with latest technologies in the field of Artificial Intelligence and Data Science.

• To provide skill-based education to master the students in problem solving and analytical skills in the area of Artificial Intelligence and Data Science.

• To develop employability skills among students in the fields of Artificial Intelligence, Data Science.

Program Educational Objectives (PEOs)

PEO1 work in the domain of Artificial Intelligence and Data Science to design ability of a computer system.

PEO2 apply analytical skills, decision making skills, leadership skills and critical thinking skills to develop Artificial Intelligence and Data Science based solutions for business problems.

PEO3 demonstrate proficiency in applying Artificial Intelligence and Data Science methodologies to solve complex real world problems across various domains such as statistics, machine learning, data analytics and Artificial Intelligence.

Program Outcomes (POs)

PO1 Engineering knowledge

Apply the knowledge of mathematics, science, Engineering fundamentals, and an Engineering specialization to the solution of complex Engineering problems.

PO2 Problem analysis

Identify, formulate, review research literature and analyze complex Engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and Engineering sciences.

PO3 Design / Development of Solutions

Design solutions for complex Engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and Environmental considerations.

PO4 Conduct Investigations of Complex Problems

Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 Modern Tool Usage

Create, select, and apply appropriate techniques, resources, and modern Engineering and IT tools including prediction and modeling to complex Engineering activities with an understanding of the limitations.

PO6 The Engineer and Society

Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practices.

PO7 Environment and Sustainability

Understand the impact of the professional Engineering solutions in societal and Environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8 Ethics

Apply ethical principles and commit to professional ethics and responsibilities and norms of Engineering practice.

PO9 Individual and Team Work

Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 Communication Skills

Communicate effectively on complex Engineering activities with the Engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 Project Management and Finance

Demonstrate knowledge and understanding of Engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary Environments.

PO12 Life-long Learning

Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Computer Laboratory – III

Course Code	Course Name	Teaching Scheme(Hrs./Week)	Credits
417534	Computer Laboratory III Computational Intelligence (417529), Distributed Computing (417530)	4	2

Course Objectives:

• To understand the fundamentals of a distributed environment in complex application.

• To introduce the concepts inspired by the human immune system and their application in problem-solving and optimization.

• To make students aware about security issues and protection mechanisms for distributed environments.

• To familiarize with various evolutionary algorithms and optimization techniques inspired by natural evolution processes.

Course Outcomes:

After completion of the course, learners should be able to-

CO1: Apply the principles on which the internet and other distributed systems are based.

CO2: Understand and apply the basic theoretical concepts and algorithms of distributed systems in problem solving.

CO3: Apply fuzzy logic techniques to model and solve problems.

CO4: Design and implement evolutionary algorithms to solve optimization and search problems in diverse domains.

CO5: Design and implement artificial immune system algorithms to solve complex problems in different domains.

Table Of Contents

Lab Assignment No.	01
Title	Design a distributed application using RPC for remote computation where client submits an integer value to the server and server calculates factorial and returns the result to the client program.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 01

Title: Design a Distributed Application Using RPC for Remote Computation of Factorial

Problem Statement: Develop a distributed application using Remote Procedure Call (RPC) where a client submits an integer value to a remote server. The server calculates the factorial of the given integer and returns the result to the client program. The application should demonstrate the concept of distributed computing by enabling remote execution of the factorial computation.

Prerequisites: Basic understanding of distributed systems, RPC mechanism, client-server architecture, factorial computation, programming in Python, and network communication fundamentals.

Software Requirements: Windows/Linux/MacOS, Python, RPC framework (XML-RPC, gRPC), IDE (VS Code, PyCharm), and networking tools.

Hardware Requirements: Intel Core i3 or higher, 4GB RAM, 100MB free disk space, and network connectivity for client-server communication.

Theory:

Introduction to Remote Procedure Call (RPC):

Remote Procedure Call (RPC) is a communication protocol that allows a program to execute procedures on a remote server as if they were local function calls. The RPC mechanism abstracts network communication complexities, making distributed computing seamless and efficient.

Working of RPC:

1. The client sends a request to the server with the necessary input data.
2. The server processes the request, executes the remote function (factorial calculation in this case), and prepares the response.
3. The server sends the computed result back to the client.
4. The client receives the result and processes it accordingly.

Factorial Computation:

The factorial of a non-negative integer **n** is the product of all positive integers from **1 to n** and is denoted as **n!**. It is defined as:

For example:

- 5! = 5 × 4 × 3 × 2 × 1 = 120
- 3! = 3 × 2 × 1 = 6
- 1! = 1
- 0! = 1 (by definition)

Implementation Approach:

To implement an RPC-based distributed factorial computation system, the following approach is used:

1. **Client Program:** Sends an integer to the server via an RPC request.
2. **Server Program:** Receives the request, computes the factorial of the given integer, and sends the result back to the client.
3. **Communication Layer:** Handles the exchange of requests and responses between the client and server using RPC mechanisms.

Advantages of RPC-Based Computation:

- Simplifies distributed computing by making remote function calls appear as local calls.
- Reduces computational load on client devices by offloading heavy calculations to the server.
- Enables modular and scalable application design.
- Facilitates resource sharing and efficient computing in networked environments.

Source Code –

Client.py

```python
import xmlrpc.client

# Create an XML-RPC client
with xmlrpc.client.ServerProxy("http://localhost:8000/RPC2") as proxy:
    try:
        # Prompt the user to enter a number
        input_value_str = input("Enter the number: ")
        input_value = int(input_value_str)

        # Call the server's calculate_factorial method
        result = proxy.calculate_factorial(input_value)
        print(f"Factorial of {input_value} is: {result}")
    except Exception as e:
        print(f"Error: {e}")
```

Output -

Enter the number: 12

Factorial of 12 is: 479001600

Server.py

```python
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.server import SimpleXMLRPCRequestHandler

class FactorialServer:
    def calculate_factorial(self, n):
        if n < 0:
            raise ValueError("Input must be a non-negative integer.")
        result = 1
        for i in range(1, n + 1):
            result *= i
        return result

# Restrict to a particular path
class RequestHandler(SimpleXMLRPCRequestHandler):
    rpc_paths = ('/RPC2',)

# Create server
with SimpleXMLRPCServer(('localhost', 8000), requestHandler=RequestHandler) as
server:
    server.register_introspection_functions()
    server.register_instance(FactorialServer())
    print("FactorialServer is ready to accept requests.")
    server.serve_forever()
```

Output - FactorialServer is ready to accept requests.

Conclusion: This practical demonstrates the efficiency of RPC in distributed computing by remotely executing factorial calculations. It highlights modular design, computational offloading, and seamless communication in client-server architectures.

Lab Assignment No.	02
Title	Design a distributed application using RMI for remote computation where client submits two strings to the server and server returns the concatenation of the given strings.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor’s Sign	

Assignment No.2

Title: Design a Distributed Application Using RMI for Remote String Concatenation

Problem Statement: Develop a distributed application using Remote Method Invocation (RMI) where a client submits two strings to a remote server. The server concatenates the given strings and returns the result to the client program. The application should demonstrate the concept of distributed computing by enabling remote execution of string concatenation.

Prerequisites: Basic understanding of distributed systems, RMI mechanism, client-server architecture, string manipulation, programming in Python, and network communication fundamentals.

Software Requirements: Windows/Linux/MacOS, Python, RPyC (Remote Python Call) library, IDE (VS Code, PyCharm), and networking tools (Wireshark, Telnet, Netcat).

Hardware Requirements: Intel Core i3 or higher, 4GB RAM, 100MB free disk space, and network connectivity for client-server communication.

Theory:

Introduction to Remote Method Invocation (RMI) in Python:

Remote Method Invocation (RMI) enables a program to call methods on remote objects hosted on another machine. In Python, this can be implemented using the **RPyC (Remote Python Call)** library, which simplifies communication between distributed applications by enabling remote execution of functions.

Working of RMI in Python:

1. The client sends a request to the server with two input strings.
2. The server processes the request, concatenates the given strings, and prepares the response.
3. The server sends the concatenated result back to the client.
4. The client receives the result and processes it accordingly.

String Concatenation:

String concatenation is the process of joining two or more strings together. In Python, it can be performed using:

- The `+` operator: `result = str1 + str2`
- The `join()` method: `result = "".join([str1, str2])`

For example:

- **Input:** "Hello" and "World"
- **Output:** "HelloWorld"

Implementation Approach:

To implement an RMI-based distributed string concatenation system in Python, the following approach is used:

1. **Client Program:** Sends two strings to the server via an RMI request.
2. **Server Program:** Receives the request, concatenates the given strings, and sends the result back to the client.
3. **Communication Layer:** Handles the exchange of requests and responses between the client and server using the RPyC library.

Advantages of RMI-Based Computation in Python:

- Simplifies distributed computing by allowing remote function execution.
- Reduces computational load on client devices by offloading processing to the server.
- Facilitates modular and scalable application design.
- Provides seamless communication between Python applications over a network.

Source Code –

Client.py

```
import Pyro4

# Get the server object using its URI
uri = input("Enter the server URI : ")
string_concatenator = Pyro4.Proxy(uri)

# Get inputs string from the user
str1 = input("Enter first string : ")
str2 = input("Enter second string : ")

# Call the remote method on the server
result = string_concatenator.concatenate(str1, str2)

print("Concatenated String : ", result)
```

Output

```
Enter the server URI :

PYRO:obj_3f70c6b68e9e4712a73a4167337c1e58@localhost:61368

Enter first string : Hello

Enter second string : World

Concatenated String :  Hello World
```

Server.py

```
import Pyro4

@Pyro4.expose
class StringConcatenator(object):
    def concatenate(self, str1, str2):
        return str1 + " " + str2

# Register the StringConcatenator class as a Pyro Object
daemon = Pyro4.Daemon()
uri = daemon.register(StringConcatenator)
```

```python
print("Server URI : ", uri)

# Start the server
daemon.requestLoop()
```

Output –

```
Server URI :  PYRO:obj_3f70c6b68e9e4712a73a4167337c1e58@localhost:61368
```

Conclusion: This practical demonstrates the efficiency of RMI in distributed computing by remotely executing string concatenation in Python. It highlights modular design, network communication, and seamless integration in client-server architectures.

Lab Assignment No.	03
Title	Implement Union, Intersection, Complement and Difference operations on fuzzy sets. Also create fuzzy relations by Cartesian product of any two fuzzy sets and perform max-min composition on any two fuzzy relations.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 03

Title: Implementation of Fuzzy Set Operations and Fuzzy Relations

Problem Statement: Implement Union, Intersection, Complement, and Difference operations on fuzzy sets. Additionally, create fuzzy relations using the Cartesian product of any two fuzzy sets and perform max-min composition on two fuzzy relations.

Prerequisites: Basic understanding of fuzzy set theory, fuzzy relations, mathematical operations on fuzzy sets, max-min composition, and Python programming.

Software Requirements: Windows/Linux/MacOS, Python (NumPy, Pandas, Matplotlib), and SciPy library for mathematical computations.

Hardware Requirements: Intel Core i3 or higher, 4GB RAM, 200MB free disk space, and an active Python environment.

Theory:

Introduction to Fuzzy Sets:

A fuzzy set is an extension of a classical set where each element has a degree of membership between 0 and 1. Fuzzy sets are used in approximate reasoning and decision-making applications.

Fuzzy Set Operations:

1. **Union (A ∪ B):** The maximum membership value for each element from both sets.
    - Formula: $\mu(A \cup B) = \max(\mu A, \mu B)$
2. **Intersection (A ∩ B):** The minimum membership value for each element from both sets.
    - Formula: $\mu(A \cap B) = \min(\mu A, \mu B)$
3. **Complement (A’):** The degree of non-membership of each element.
    - Formula: $\mu(A') = 1 - \mu(A)$
4. **Difference (A - B):** Elements that belong to A but not to B.
    - Formula: $\mu(A - B) = \min(\mu A, 1 - \mu B)$

Fuzzy Relations and Cartesian Product:

A fuzzy relation is a mapping between two fuzzy sets using the Cartesian product.

- If A and B are two fuzzy sets, their Cartesian product results in a fuzzy relation R.
- The membership function of the relation is defined as:
    - $\mu R(A, B) = \min(\mu A(x), \mu B(y))$

Max-Min Composition of Fuzzy Relations:

- The max-min composition is used to derive a new fuzzy relation from two existing fuzzy relations.
- Formula:
    - $\mu R \circ S(x, z) = \text{max\_y} (\min(\mu R(x, y), \mu S(y, z)))$

Implementation Approach:

1. **Define fuzzy sets:** Represent them as Python dictionaries or arrays.
2. **Perform operations:** Compute union, intersection, complement, and difference using NumPy functions.
3. **Create fuzzy relations:** Use the Cartesian product to form a relation matrix.
4. **Max-Min Composition:** Implement a function to compute max-min composition between two fuzzy relations.
5. **Visualization:** Use Matplotlib to plot fuzzy sets and relations.

Applications of Fuzzy Sets and Relations:

- Fuzzy logic in artificial intelligence and expert systems.
- Decision-making in uncertain environments.
- Control systems such as washing machines and air conditioning.
- Image processing and pattern recognition.

Source Code –

```
import numpy as np

# Function to perform Union operation on fuzzy sets

def fuzzy_union(A, B): return np.maximum(A, B)

# Function to perform Intersection operation on fuzzy sets

def fuzzy_intersection(A, B): return np.minimum(A, B)

# Function to perform Complement operation on a fuzzy set

def fuzzy_complement(A): return 1 - A

# Function to perform Difference operation on fuzzy sets

def fuzzy_difference(A, B): return np.maximum(A, 1 - B)

# Function to create fuzzy relation by Cartesian product of two fuzzy sets

def cartesian_product(A, B): return np.outer(A, B)

# Function to perform Max-Min composition on two fuzzy relations

def max_min_composition(R, S): return np.max(np.minimum.outer(R, S), axis=1)

# Example usage

A = np.array([0.2, 0.4, 0.6, 0.8]) # Fuzzy set A

 B = np.array([0.3, 0.5, 0.7, 0.9])# Fuzzy set B

# Operations on fuzzy sets

union_result = fuzzy_union(A, B)

intersection_result = fuzzy_intersection(A, B)

complement_A = fuzzy_complement(A)

difference_result = fuzzy_difference(A, B)

print("Union:", union_result)

print("Intersection:", intersection_result)

print("Complement of A:", complement_A)
print("Difference:", difference_result)

# Fuzzy relations

R = np.array([0.2, 0.5, 0.4]) # Fuzzy relation R

S = np.array([0.6, 0.3, 0.7]) # Fuzzy relation S

 # Cartesian product of fuzzy relations

cartesian_result = cartesian_product(R, S)

# Max-Min composition of fuzzy relations

composition_result = max_min_composition(R, S)

print("Cartesian product of R and S:") print(cartesian_result)

 print("Max-Min composition of R and S:")

print(composition_result)
```

Conclusion: This practical demonstrates the implementation of fuzzy set operations and fuzzy relations using Python. It highlights the importance of fuzzy logic in decision-making and real-world applications.

Lab Assignment No.	04
Title	Write code to simulate requests coming from clients and distribute them among the servers using the load balancing algorithms.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor’s Sign	

Assignment No. 04

Title: Simulation of Client Requests and Load Distribution Using Load Balancing Algorithms

Problem Statement: To design and implement a simulation of client requests and distribute these requests among multiple servers using various load balancing algorithms such as Round Robin, Least Connections, and Random Selection in Jupyter Notebook.

Prerequisites:

- Basic knowledge of Python programming
- Understanding of networking concepts
- Familiarity with load balancing algorithms
- Basic knowledge of Jupyter Notebook environment

Software Requirements:

- Jupyter Notebook
- Python 3.x

Hardware Requirements:

- A computer with minimum 4 GB RAM
- Dual-core processor or higher
- Stable internet connection

Theory: Load balancing is a technique used to distribute network or application traffic across multiple servers. This ensures no single server bears too much demand, enhancing responsiveness and availability of applications.

Load balancing can be broadly categorized into two types:

1. **Static Load Balancing:** In this method, the distribution of traffic is predefined and does not change based on server conditions. It works well for systems with uniform load distribution.
2. **Dynamic Load Balancing:** This method adapts to changing conditions in real-time, distributing traffic based on current server load, response time, or other metrics.

Common Load Balancing Algorithms:

1. **Round Robin:** Requests are distributed cyclically to all servers. This is simple to implement and works well when all servers have similar capabilities.
2. **Least Connections:** Requests are sent to the server with the fewest active connections. This algorithm is efficient in environments where requests have varying processing times.
3. **Random Selection:** Requests are assigned to a randomly selected server. It is easy to implement but may lead to uneven load distribution if not managed properly.

Importance of Load Balancing:

- Enhances system reliability and availability
- Improves application performance
- Reduces server downtime
- Efficient utilization of resources

Source Code –

```
import random

 import time

 class   Server:

       def__init__(self,   id):

    self.id=id

      self.current_connections   = 0

     def  handle_request(self):

            """Simulate handling  a request by a server."""

                        self.current_connections + = 1

print(f"Server {self.id}  is  handling  the  request.  Total  connections:
            {self.current_connections}")

            def release_request(self):

            """Release a request from the server."""
            If self.current_connections > 0:

            self.current_connections -= 1

      print(f"Server  {self.id}has released a request. Total connections
{self.current_connections}")

 class   LoadBalancer:

   def__init__(self, servers):

      self.servers = servers

      self.server_count = len(servers)

      self.round_robin_index = 0

   def  round_robin(self):

    """Round  Robin  load     balancing algorithm."""

      server  = self.servers[self.round_robin_index]

      self.round_robin_index = (self.round_robin_index + 1)% self.server_count

      return   server

    def  least_connections(self):

      """Least Connections  load balancing algorithm."""

         Server = min(self.servers,  key=lambda s: s.current_connections)

         return server

    def random_selection(self):

   """Random load balancing algorithm."""

      return   random.choice(self.servers)

      def distribute_request(self,     algorithm="round_robin"):

"""Distribute   requests among servers based on the selected algorithm."""

      If algorithm = = "round_robin":


      server  = self.round_robin()
```

```
elif algorithm = ="least_connections":
          server = self.least_connections()
elif algorithm = = "random":
         server = self.random_selection()
else:
    raise ValueError("Unknown algorithm")
            server.handle_request()
 # Simulate the scenario
 def simulate_requests():
            servers = [Server(i) for i in range(1,   4)]       # Three servers
            load_balancer  = LoadBalancer(servers)
            # Simulate 10  client   requests
            for_in  range(10):
            algorithm = random.choice(["round_robin",  "least_connections",    "random"])
            # Randomly select an algorithm
            print(f"\nDistributing  request using    {algorithm} algorithm.")
            load_balancer.distribute_request(algorithm)
            time.sleep(0.5)            # Simulate time delay between requests
            # Simulate releasing requests
            For server in servers:
                         server.release_request()
 # Run the simulation
 simulate_requests()
```

Conclusion - Through this practical, we have explored the fundamental concepts of load balancing and its significance in distributed systems. By simulating client requests and implementing various load balancing algorithms such as Round Robin, Least Connections, and Random Selection, we observed how different strategies impact the distribution of traffic among servers.

Lab Assignment No.	05
Title	Optimization of genetic algorithm parameter in hybrid genetic algorithm-neural network modelling: Application to spray drying of coconut milk.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 05

Title: Optimization of Genetic Algorithm Parameters in Hybrid Genetic Algorithm-Neural Network Modelling: Application to Spray Drying of Coconut Milk

Problem Statement: Optimize the parameters of a genetic algorithm in a hybrid genetic algorithm-neural network (GA-NN) model to enhance the accuracy and efficiency of spray drying process optimization for coconut milk.

Prerequisites: Basic knowledge of genetic algorithms, neural networks, hybrid optimization techniques, machine learning, and Python programming. Familiarity with Jupyter Notebook and scientific computation libraries such as NumPy, TensorFlow, and SciPy.

Software Requirements: Windows/Linux/MacOS, Python (NumPy, TensorFlow/Keras, SciPy, Matplotlib), Jupyter Notebook, and Scikit-learn for model evaluation.

Hardware Requirements: Intel Core i5 or higher, 8GB RAM, 500MB free disk space, and an active Python environment.

Theory:

Introduction to Genetic Algorithm (GA) and Neural Networks (NN):

Genetic Algorithm (GA) is an optimization technique inspired by natural selection, commonly used to find optimal solutions in search spaces. Neural Networks (NN) are machine learning models designed to recognize patterns and relationships in data. When combined, GA optimizes the hyperparameters of NN models to improve their performance.

Hybrid Genetic Algorithm-Neural Network Modelling:

1. **Genetic Algorithm (GA) in Optimization:**
   - GA works by evolving a population of candidate solutions using selection, crossover, and mutation operations.
   - The objective function evaluates model performance, and the best individuals are selected for the next generation.
2. **Neural Network (NN) for Spray Drying Modelling:**

  - NN models predict outcomes in spray drying processes based on input parameters such as temperature, air velocity, and feed rate.
  - Optimizing NN hyperparameters (e.g., learning rate, number of layers, number of neurons) enhances predictive accuracy.

Optimization Process in GA-NN:

1. **Encoding:** Represent NN hyperparameters as a chromosome.
2. **Fitness Evaluation:** Train and validate the NN using different parameter sets, evaluating performance based on error metrics.
3. **Selection:** Choose the best-performing individuals.
4. **Crossover & Mutation:** Generate new candidates by combining and slightly modifying existing parameters.
5. **Iteration:** Repeat the process until the optimal parameter set is found.

Application to Spray Drying of Coconut Milk:

- Spray drying is a process used to convert liquid coconut milk into powder form.
- The optimization of process parameters (e.g., inlet air temperature, feed concentration) is crucial for maximizing yield and quality.
- The GA-NN model improves predictive capability, enabling better control over drying conditions.

Implementation Approach:

1. **Data Collection:** Gather experimental spray drying data.
2. **Preprocessing:** Normalize and split data for training/testing.
3. **GA Implementation:** Initialize and evolve a population of NN hyperparameter sets.
4. **Train & Evaluate NN:** Use optimized parameters to train the NN.
5. **Performance Analysis:** Compare GA-optimized NN results with standard models.
6. **Visualization:** Use Matplotlib to plot optimization trends and results.

Applications of GA-NN Hybrid Models:

- Optimization of industrial processes.
- Predictive modeling in food processing.

- AI-driven quality control in manufacturing.
- Adaptive control systems for process automation.

Source Code –

```
# Import Required Libraries

import numpy as np import pandas as pd

import tensorflow as tf from tensorflow.keras.models

import Sequential from tensorflow.keras.layers

import Dense from sklearn.model_selection

import train_test_split from sklearn.metrics

import mean_squared_error from deap

import base, creator, tools, algorithms

import random import matplotlib.pyplot as plt

import warnings warnings.filterwarnings("ignore", category=UserWarning)

# Load and Preprocess Data

# Simulate dataset (replace with real-world data if available)

np.random.seed(42)

 n_samples = 100

X = np.random.uniform(low=30, high=100, size=(n_samples, 2)) # e.g., temperature, feed rate

y = 0.8 * X[:, 0] - 0.5 * X[:, 1] + np.random.normal(0, 5, n_samples) # Spray drying efficiency

# Split the dataset

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Normalize inputs

X_mean, X_std = X_train.mean(axis=0), X_train.std(axis=0)

X_train = (X_train - X_mean) / X_std

 X_test = (X_test - X_mean) / X_std
```

```
# Define Neural Network

def build_nn(num_neurons, learning_rate):

model = Sequential([ Dense(num_neurons, activation='relu', input_shape=(X_train.shape[1],)), Dense(1) ])

model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=learning_rate), loss='mse')

return model

# Genetic Algorithm for Optimization

# Define GA to optimize number of neurons and learning rate

def evaluate_fitness(individual):

num_neurons = int(individual[0]) # Number of neurons

learning_rate = individual[1] # Lerning rate

model = build_nn(num_neurons, learning_rate)

model.fit(X_train, y_train, epochs=20, verbose=0, batch_size=10)

# Predict and calculate mean squared error on test set

y_pred = model.predict(X_test, verbose=0)

mse = mean_squared_error(y_test, y_pred)

 return mse,

 # GA Configuration

toolbox = base.Toolbox()

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))# Minimize MSE

creator.create("Individual", list, fitness=creator.FitnessMin)

toolbox.register("attr_num_neurons", random.randint, 5, 50) # Range for neurons
toolbox.register("attr_learning_rate", random.uniform, 0.001, 0.01) # Range for learning rate
toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.attr_num_neurons, toolbox.attr_learning_rate), n=1)

toolbox.register("population", tools.initRepeat, list, toolbox.individual)

toolbox.register("mate", tools.cxBlend, alpha=0.5)
toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=0.1, indpb=0.2)

toolbox.register("select",       tools.selTournament,       tournsize=3)       toolbox.register("evaluate",
evaluate_fitness)

# Run GA

random.seed(42)

population = toolbox.population(n=10)

ngen = 20

cxpb, mutpb = 0.5, 0.2

result,   log   =   algorithms.eaSimple(population,   toolbox,   cxpb=cxpb,   mutpb=mutpb,   ngen=ngen,
verbose=True)

#          Best      Solution

 best_individual =tools.selBest(population,k=1)[0]

 print(f"BestIndividual:   Neurons={best_individual[0]},    LearningRate={best_individual[1]}")

 #Train   and evaluate the final model

 final_model =build_nn(int(best_individual[0]),   best_individual[1])

 final_model.fit(X_train, y_train, epochs=50, verbose=1,  batch_size=10)

 #Test the model

 y_pred = final_model.predict(X_test)

 final_mse = mean_squared_error(y_test, y_pred)

 print(f"Final MSE on Test Data:  {final_mse}"

# Visualize Results

 # Plot true vs predicted values

 plt.scatter(y_test, y_pred, c='blue', label='Predictions')

 plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)],    color='red', linestyle='--')

 plt.xlabel("True Values")

 plt.ylabel("Predicted Values")
 plt.legend()

 plt.title("True vs Predicted        Values")

 plt.show()
```

Conclusion: This practical demonstrates the application of a hybrid GA-NN model for optimizing genetic algorithm parameters in spray drying of coconut milk. The approach enhances predictive accuracy and improves process efficiency through AI-driven optimization techniques.

Lab Assignment No.	06
Title	Implementation of Clonal selection algorithm using Python.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 06

Title: Implementation of Clonal Selection Algorithm Using Python

Problem Statement: To design and implement the Clonal Selection Algorithm (CSA) using Python in Jupyter Notebook to optimize problem-solving capabilities inspired by the natural immune system.

Prerequisites:

- Basic knowledge of Python programming
- Understanding of optimization algorithms
- Familiarity with bio-inspired algorithms
- Basic knowledge of Jupyter Notebook environment

Software Requirements:

- Jupyter Notebook
- Python 3.x

Hardware Requirements:

- A computer with minimum 4 GB RAM
- Dual-core processor or higher
- Stable internet connection

Theory: The Clonal Selection Algorithm (CSA) is a bio-inspired optimization technique based on the clonal selection principle of the adaptive immune system. It is used to solve complex optimization problems by mimicking the process of natural selection, cloning, and mutation of immune cells.

Key Concepts:

1. **Antigen Recognition:** Identifying candidate solutions (antibodies) that can respond to the problem (antigen).
2. **Cloning:** Creating multiple copies of high-affinity antibodies.

3. **Mutation:** Introducing variations in cloned antibodies to explore new solutions.
4. **Selection:** Retaining the best solutions based on affinity or fitness.

The Clonal Selection Algorithm operates in an iterative manner, improving the solution population over successive generations. The fundamental processes are inspired by the natural immune response mechanism, which efficiently identifies and eliminates pathogens.

1. **Affinity Maturation:** The algorithm continuously improves the quality of solutions through the process of affinity maturation, where high-affinity antibodies are selected, cloned, and subjected to hypermutation to enhance their performance.
2. **Hypermutation:** This process introduces diversity into the population by applying mutations at higher rates, especially to low-affinity antibodies, allowing the exploration of new regions in the solution space.
3. **Replacement Strategy:** CSA employs a replacement mechanism where poorly performing antibodies are eliminated and replaced with new randomly generated antibodies. This prevents premature convergence and maintains population diversity.
4. **Convergence Criteria:** The algorithm terminates when a stopping condition is met, such as reaching a maximum number of generations or achieving a satisfactory fitness level.

Mathematical Model: The mathematical representation of CSA involves:

- **Affinity Calculation:** , where is the fitness function to be minimized.
- **Selection Probability:** Proportional to the affinity score, determining the likelihood of an antibody being cloned.
- **Mutation Rate:** Inversely proportional to the affinity, i.e., higher affinity leads to lower mutation rates and vice versa.

Applications of CSA:

- Optimization problems
- Pattern recognition
- Machine learning
- Data mining
- Network security
- Robotics path planning

Source Code –

```
# Import Required Libraries

import numpy as np

import matplotlib.pyplot as plt

# Define the Objective Function

def     objective_function(x):

   return x**2 + 3*x + 5

# Initialize Population

def initialize_population(size,lower_bound,  upper_bound):

  return np.random.uniform(lower_bound, upper_bound, size)

# Evaluate Affinity (Fitness)

def evaluate_fitness(population):

            return  1 / (1 + objective_function(population))

# Select Best Antibodies

def select_best(population, fitness, num_best):

            sorted_indices = np.argsort(-fitness)

            return  population[sorted_indices[:num_best]]

# Clone Selected Antibodies

def     clone_population(best_population,    num_clones):

            clones  = np.repeat(best_population,  num_clones)

return  clones

# Hypermutation (Random Mutation)

def mutate(clones, mutation_rate):

mutation = np.random.uniform(-mutation_rate, mutation_rate, size=clones.shape)

            return  clones  + mutation
```

```
# Replace Worst Solutions

def replace_worst(population,new_population, num_replace):

        population[-num_replace:] = new_population[:num_replace]

          return population

# Implement the Clonal Selection Algorithm

def clonal_selection_algorithm(pop_size=20, generations=50,num_best=5,num_clones=3,
        mutation_rate=0.1):

        lower_bound,  upper_bound   = -10,   10

population = initialize_population(pop_size, lower_bound,  upper_bound)

    best_fitness_history = []

        for generation  in range(generations):

                fitness  = evaluate_fitness(population)

        best_population = select_best(population, fitness,     num_best)

                clones  = clone_population(best_population, num_clones)

        mutated_clones = mutate(clones, mutation_rate)

        fitness_mutated = evaluate_fitness(mutated_clones)

        best_mutated   = select_best(mutated_clones, fitness_mutated, num_best)

                population = replace_worst(population, best_mutated, num_best)

        best_fitness_history.append(np.max(fitness))

                if generation % 10 = = 0:

                print(f"Generation {generation}: Best fitness = {np.max(fitness)}")

        return  best_fitness_history,   population

# Run the Algorithm

best_fitness,    final_population = clonal_selection_algorithm()

# Plot the Results

plt.plot(best_fitness)
plt.xlabel('Generations')
plt.ylabel('Best Fitness')
plt.title('Clonal Selection Algorithm Performance')
plt.show()
```

Conclusion - Through this practical, we have explored the Clonal Selection Algorithm and its role in solving optimization problems inspired by the natural immune system. This practical highlighted the efficiency of bio-inspired algorithms in addressing complex problem-solving tasks, demonstrating the algorithm's ability to find optimal solutions through adaptive processes.

Lab Assignment No.	07
Title	To apply the artificial immune pattern recognition to perform a task of structure damage Classification.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 07

Title: Application of Artificial Immune Pattern Recognition for Structural Damage Classification

Problem Statement: To apply artificial immune pattern recognition techniques to perform the task of structural damage classification, utilizing bio-inspired algorithms to detect and classify damage in structural components.

Prerequisites:

- Basic knowledge of Python programming
- Understanding of pattern recognition and classification techniques
- Familiarity with bio-inspired algorithms and artificial immune systems
- Basic knowledge of Jupyter Notebook environment

Software Requirements:

- Jupyter Notebook
- Python 3.x
- Required Python libraries: NumPy, Pandas, Scikit-learn, Matplotlib

Hardware Requirements:

- A computer with minimum 4 GB RAM
- Dual-core processor or higher
- Stable internet connection

Theory: Artificial Immune Systems (AIS) are computational models inspired by the principles and processes of the biological immune system. AIS are particularly effective in pattern recognition tasks, such as structural damage classification, due to their adaptive learning capabilities, robustness, and efficiency in handling complex, dynamic data environments.

The AIS framework operates by simulating the immune system's mechanisms of recognizing, learning, and remembering patterns. In the context of structural damage classification, AIS mimics the immune system’s ability to detect and classify anomalies (damages) based on extracted structural features. The system utilizes concepts such as antigen-antibody interactions, clonal selection, and affinity maturation to optimize classification performance.

Key Concepts:

1. **Pattern Recognition:** The process of identifying and categorizing patterns within structural health monitoring data to detect potential damage.
2. **Antigen-Antibody Interaction:** Structural features representing potential damage (antigens) are compared with candidate solutions (antibodies) generated by the AIS model.
3. **Clonal Selection Principle:** High-affinity antibodies (accurate classifiers) are selected, cloned, and mutated to improve the system's ability to classify structural conditions.
4. **Affinity Measure:** A quantitative assessment of how well an antibody (classifier) recognizes and classifies a given antigen (damage pattern).
5. **Mutation and Diversity:** Introduces variability within cloned antibodies to explore new solutions and improve generalization capability.
6. **Memory Set:** A collection of the best-performing antibodies that are retained for future classification tasks, enhancing the system’s learning over time.

Mathematical Model:

- **Affinity Calculation:** Measures the similarity between structural features (antigens) and candidate solutions (antibodies) using distance metrics or similarity functions.
- **Classification Rule:** Assigns structural conditions (e.g., damaged or undamaged) based on the highest affinity score.
- **Mutation Operator:** Applies controlled alterations to cloned antibodies to maintain diversity and prevent overfitting.

Applications:

- Structural health monitoring in civil engineering
- Damage detection in bridges, buildings, and aerospace structures
- Mechanical component fault diagnosis
- Monitoring and maintenance planning in critical infrastructure systems

Source Code –

```
# Import Required Libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns from sklearn.model_selection
import train_test_split from sklearn.metrics
import accuracy_score, confusion_matrix, classification_report
#Load and Prepare the Dataset
df = pd.read_csv(r"C:\Users\saira\Downloads\structural_damage_data.csv")
df.head()
df.tail()
df.isnull().sum()
# Feature Selection and Spiltting data
X = df.drop('Damage_Level', axis=1) y = df['Damage_Level']
# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Artificial Immune Pattern Recognition (AIPR) Algorithm
class AIPR:
def __init__(self, n_clones=20, mutation_rate=0.05, generations=100):
 self.n_clones = n_clones
self.mutation_rate = mutation_rate
self.generations = generations
def fit(self, X, y):
self.memory_cells = np.random.rand(self.n_clones, X.shape[1])
self.labels = np.random.choice(np.unique(y), self.n_clones)

for _ in range(self.generations):

 for i in range(self.n_clones):

clone = self.memory_cells[i] + self.mutation_rate * np.random.randn(X.shape[1])

 clone_fitness = self._fitness(clone, X, y)

if clone_fitness > self._fitness(self.memory_cells[i], X, y):

self.memory_cells[i] = clone

 def predict(self, X):

preds = []

for x in X:

distances = np.linalg.norm(self.memory_cells - x, axis=1)

 pred = self.labels[np.argmin(distances)]

preds.append(pred)

return np.array(preds)

def _fitness(self, antibody, X, y):

distances = np.linalg.norm(self.memory_cells - antibody, axis=1)

closest_label = self.labels[np.argmin(distances)]

return np.mean(closest_label == y)

# Train and Evaluate the Model

# Initialize and train the AIPR model

model = AIPR(n_clones=20, mutation_rate=0.05, generations=100)

model.fit(X_train.values, y_train.values)

# Make predictions

y_pred = model.predict(X_test.values)

# Evaluation
```

```
print("Accuracy:", accuracy_score(y_test, y_pred))

print("Classification Report:\n", classification_report(y_test, y_pred))

# Confusion Matrix

sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, cmap='Blues')

 plt.title('Confusion Matrix')

plt.show()

# Visualize Result

# Compare true vs predicted damage levels

plt.figure(figsize=(10, 5))

 plt.plot(y_test.values, label='True')

 plt.plot(y_pred, label='Predicted', alpha=0.7)

plt.legend()

plt.title('True vs Predicted Damage Levels')

plt.show()
```

Conclusion: Through this practical, we have explored the application of artificial immune pattern recognition for structural damage classification. By implementing bio-inspired algorithms in Python, we demonstrated the effectiveness of AIS in identifying and classifying structural damage patterns. This practical highlighted the potential of artificial immune systems in enhancing structural health monitoring and improving the reliability of damage detection processes.

Lab Assignment No.	08
Title	Implement DEAP (Distributed Evolutionary Algorithms) using Python.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor’s Sign	

Assignment No. 08

Title: Implementation of DEAP (Distributed Evolutionary Algorithms in Python) Using Python

Problem Statement: To design and implement Distributed Evolutionary Algorithms (DEAP) using Python in Jupyter Notebook for solving complex optimization problems through evolutionary computation techniques.

Prerequisites:

- Basic knowledge of Python programming
- Understanding of evolutionary algorithms and optimization techniques
- Familiarity with genetic algorithms and related bio-inspired methods
- Basic knowledge of Jupyter Notebook environment

Software Requirements:

- Jupyter Notebook
- Python 3.x
- Required Python libraries: DEAP, NumPy, Matplotlib (for visualization)

Hardware Requirements:

- A computer with minimum 4 GB RAM
- Dual-core processor or higher
- Stable internet connection (optional for additional resources)

Theory: The Distributed Evolutionary Algorithms in Python (DEAP) framework is an advanced library designed to facilitate the rapid prototyping and implementation of evolutionary algorithms. DEAP supports a wide range of bio-inspired algorithms, including genetic algorithms, genetic programming, evolution strategies, and more. It is highly flexible, allowing users to customize the algorithm components to address specific problem domains.

Key Concepts:

1. **Evolutionary Algorithms (EAs):** A family of optimization algorithms inspired by the process of natural selection, involving operations like selection, crossover, and mutation.

2. **Genetic Algorithms (GAs):** A type of EA where candidate solutions (individuals) evolve over generations to find optimal or near-optimal solutions.
3. **Selection Mechanism:** Identifies the fittest individuals based on a fitness function to participate in reproduction.
4. **Crossover (Recombination):** Combines genetic information from two parent individuals to create offspring with mixed characteristics.
5. **Mutation:** Introduces random changes to individual solutions, promoting genetic diversity and preventing premature convergence.
6. **Fitness Evaluation:** Measures the quality of each candidate solution based on a problem-specific objective function.

Mathematical Model:

- **Fitness Function:**
- **Selection Probability:** Based on fitness proportionate selection or tournament selection methods.
- **Crossover Rate (Pc):** Defines the likelihood of performing crossover between parent individuals.
- **Mutation Rate (Pm):** Determines the probability of applying random mutations to offspring.

Applications:

- Optimization problems in engineering and science
- Machine learning model tuning
- Scheduling and resource allocation
- Robotics path planning
- Game strategy optimization

Source Code –

```
# Import Required Libraries

import random from deap

import base, creator, tools, algorithms
import numpy as np

# Define the evaluation function (minimize a simple mathematical function)

def eval_func(individual):

# Example evaluation function (minimize a quadratic function)

return sum(x ** 2 for x in individual),

 # DEAP setup

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

creator.create("Individual", list, fitness=creator.FitnessMin)

 toolbox = base.Toolbox()

# Define attributes and individuals

 toolbox.register("attr_float", random.uniform, -5.0, 5.0) # Example: Float values between -5 and
5

toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_float, n=3) #
Example: 3-dimension

toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Evaluation function and genetic operators

toolbox.register("evaluate", eval_func)

toolbox.register("mate", tools.cxBlend, alpha=0.5)

toolbox.register("mutate", tools.mutGaussian, mu=0, sigma=1, indpb=0.2)
toolbox.register("select", tools.selTournament, tournsize=3)

# Create population

 population = toolbox.population(n=50)

 # Genetic Algorithm

parameters generations = 20

 # Run the algorithm

for gen in range(generations):

offspring = algorithms.varAnd(population, toolbox, cxpb=0.5, mutpb=0.1)
```

```
fits = toolbox.map(toolbox.evaluate, offspring)

for fit, ind in zip(fits, offspring):

ind.fitness.values = fit

population = toolbox.select(offspring, k=len(population))

# Get the best individual after generations

best_ind = tools.selBest(population, k=1)[0]

 best_fitness = best_ind.fitness.values[0]

print("Best individual:", best_ind)

print("Best fitness:", best_fitness)
```

Conclusion: Through this practical, we have explored the implementation of Distributed Evolutionary Algorithms using the DEAP library in Python. By leveraging evolutionary principles such as selection, crossover, and mutation, we demonstrated the effectiveness of DEAP in solving complex optimization problems. This practical highlighted the flexibility and power of DEAP in developing custom evolutionary algorithms for diverse applications.

Lab Assignment No.	09
Title	Implement Ant colony optimization by solving the Traveling salesman problem using python Problem statement- A salesman needs to visit a set of cities exactly once and return to the original city. The task is to find the shortest possible route that the salesman can take to visit all the cities and return to the starting city.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 09

Title: Implementation of Ant Colony Optimization for Solving the Traveling Salesman Problem Using Python

Problem Statement: A salesman needs to visit a set of cities exactly once and return to the original city. The task is to find the shortest possible route that the salesman can take to visit all the cities and return to the starting city, using Ant Colony Optimization (ACO) techniques.

Prerequisites:

- Basic knowledge of Python programming
- Understanding of optimization techniques and combinatorial problems
- Familiarity with metaheuristic algorithms, specifically Ant Colony Optimization
- Basic knowledge of Jupyter Notebook environment

Software Requirements:

- Jupyter Notebook
- Python 3.x
- Required Python libraries: NumPy, Matplotlib, NetworkX (for visualization of paths)

Hardware Requirements:

- A computer with minimum 4 GB RAM
- Dual-core processor or higher
- Stable internet connection

Theory: Ant Colony Optimization (ACO) is a probabilistic technique inspired by the foraging behavior of ants in nature. Ants find the shortest path between their colony and food sources by depositing pheromones along their trails. Over time, shorter paths accumulate more pheromones, attracting more ants and reinforcing the optimal route.

In the context of the Traveling Salesman Problem (TSP), ACO is used to simulate the behavior of artificial ants that explore different routes between cities. These ants communicate indirectly through pheromone trails, which guide the search process towards optimal or near-optimal solutions.

Key Concepts:

1. **Pheromone Trail (τ):** A numerical value representing the desirability of a path between two cities. Higher pheromone levels indicate preferred routes.
2. **Heuristic Information (η):** Represents the visibility or attractiveness of a path, often inversely proportional to the distance between cities.
3. **Probability Transition Rule:** Determines the likelihood of an ant choosing a specific path based on pheromone concentration and heuristic information.
4. **Pheromone Update Rule:** Involves evaporation (reducing pheromone intensity over time) and deposition (adding pheromones based on the quality of the solution).
5. **Exploration vs. Exploitation:** Balances between exploring new paths and exploiting known good routes to optimize the solution.

Applications:

- Routing and logistics optimization
- Network design and resource allocation
- Scheduling problems
- Vehicle routing in transportation systems

Advantages of ACO:

- Efficient for large-scale problems
- Adaptive to dynamic environments
- Balances exploration and exploitation

Challenges:

- Sensitive to parameter tuning
- Risk of premature convergence
- Pathfinding in robotics and AI

Source Code –

```
# Import Required Libraries
 importnumpy as np
 importrandom
 importmatplotlib.pyplot as plt
 # Define coordinates  for each city (for visualization purposes)
 city_coordinates = np.array([
                                        [0, 0],          # City  0
                                        [10, 0],         # City  1
                                        [10, 10],        # City  2
                                        [0, 10]          # City 3
 ])
 # Define the distance matrix (distances between cities)
distance_matrix = np.array([
                                        [0,     10,     15,     20],
                                        [10,    0,      35,     25],
                                        [15,    35,     0,      30],
                                        [20,    25,     30,     0]
 ])
 # Parameters  for Ant Colony Optimization
 num_ants = 10
 num_iterations = 50
 evaporation_rate = 0.5
 pheromone_constant  = 1.0
 heuristic_constant = 1.0
 # Initialize pheromone matrix and visibilitymatrix

 num_cities = len(distance_matrix)

 pheromone = np.ones((num_cities,   num_cities))    # Pheromone  matrix

 # Handle division by zero in visibility matrix (replace 0s with infinity)

 Visibility = np.where(distance_matrix = = 0, np.inf,1 / distance_matrix)

# ACO algorithm

 For iteration in range(num_iterations):

                                        ant_routes = []

                                        for ant in range(num_ants):

           current_city = random.randint(0, num_cities - 1)

                              visited_cities   = [current_city]

                              route = [current_city]

           while   len(visited_cities) < num_cities:

                         probabilities = []

          for city  in range(num_cities):

                              if city not in visited_cities:

                     pheromone_value = pheromone[current_city][city]

                     visibility_value = visibility[current_city][city]

       probability   =   (pheromone_value   **   pheromone_constant)   *   (visibility_value   **
           heuristic_constant

           probabilities.append((city, probability))

           probabilities    = sorted(probabilities, key=lambda    x: x[1],reverse=True)

           selected_city   = probabilities[0][0]

           route.append(selected_city)

           visited_cities.append(selected_city)

           current_city = selected_city
```

```
   ant_routes.append(route)

# Update pheromone levels

delta_pheromone = np.zeros((num_cities, num_cities))

    for  ant, route in enumerate(ant_routes):

  for I in range(len(route) - 1):
         city_a  = route[i]

         city_b  = route[I +1]

delta_pheromone[city_a][city_b] +=  1 / distance_matrix[city_a][city_b]

 delta_pheromone[city_b][city_a] += 1/distance_matrix[city_a][city_b]

 pheromone = (1 -evaporation_rate) * pheromone + delta_pheromone

# Find  the best route

 best_route_index   =   np.argmax([sum(distance_matrix[cities[i]][cities[(I   +1)%   num_cities]]
         for I in range(num_cities

 best_route = ant_routes[best_route_index]

 shortest_distance = sum(distance_matrix[best_route[i]][best_route[(I +1)  %         num_cities]]
         for I in range(num_cities

print("Best route:", best_route)

 print("Shortest distance:", shortest_distance)

# Visualize the best route using city coordinates

best_route_coords = city_coordinates[best_route] # Get the coordinates of the best route

best_route_coords = np.vstack([best_route_coords, best_route_coords[0]]) # Add the starting
city at the end to close

x, y = zip(*best_route_coords) # Unpack coordinates

 plt.plot(x, y, 'o-', color='blue')

 plt.title(f"Best Route with Length: {shortest_distance}")

 plt.xlabel('X')

plt.ylabel('Y')
plt.show()
```

Conclusion: Ant Colony Optimization provides an effective, nature-inspired approach to solving the Traveling Salesman Problem. By simulating the behavior of ants and leveraging pheromone-based communication, ACO efficiently explores possible routes, balancing exploration of new paths and exploitation of known good paths. The algorithm's flexibility and robustness make it suitable for complex optimization problems beyond TSP.

Lab Assignment No.	10
Title	Create and Art with Neural style transfer on given image using deep learning.
Roll No.	
Class	BE AI & DS
Date Of Completion	
Subject	Computer Laboratory III[417534]
Assessment Marks	
Assessor's Sign	

Assignment No. 10

Title: Art Creation Using Neural Style Transfer with Deep Learning

Problem Statement: Neural Style Transfer (NST) is a deep learning technique used to combine the content of one image with the artistic style of another. This practical introduces the concept of NST and demonstrates how it can be implemented using a pre-trained deep learning model.

Prerequisites:

- Basic understanding of Convolutional Neural Networks (CNNs)
- Knowledge of Python programming
- Familiarity with deep learning frameworks like **TensorFlow** or **PyTorch**
- Understanding of the concept of **content** and **style** in images

Software Requirements:

- Python 3.x
- TensorFlow or PyTorch (for implementing Neural Style Transfer)
- Jupyter Notebook (or any Python IDE)
- Libraries: NumPy, Matplotlib, Keras, OpenCV, etc.

Hardware Requirements:

- A system with at least 8GB of RAM
- A Graphics Processing Unit (GPU) is recommended for faster computation (optional)
- Disk space for image datasets

Theory:

Neural Style Transfer (NST) is a technique in deep learning that allows the combination of the content of one image with the artistic style of another. The aim of NST is to create a new image that retains the structure and objects (content) of the source image, while adopting the visual appearance, color patterns, textures, and artistic style of the style image.

The underlying concept of NST can be explained through two primary concepts: **content** and **style**.

1. **Content**:
   Content refers to the key structures and objects present in an image. In the context of NST, the content of an image is typically related to the arrangement of objects and their spatial relationships. For example, a portrait may contain content such as the shape of the face, the location of eyes, nose, and mouth, and other objects present in the scene.
2. **Style**:
   Style refers to the visual appearance, textures, colors, and patterns found in an image. The style can be defined by elements like color schemes, textures, brush strokes, and patterns. In art, style often characterizes the work of specific artists (e.g., Van Gogh's swirling brushstrokes or Picasso's abstract shapes).

How Neural Style Transfer Works:

NST uses deep convolutional neural networks (CNNs) to separate the content and style of an image. A pre-trained network, such as **VGG-19**, is employed to extract deep features from both the content and style images.

1. **ContentLoss**:
   To retain the content of the original image, the difference between the content features of the generated image and the original content image is measured. This difference is referred to as **content loss**, which is minimized during the optimization process.
2. **StyleLoss**:
   To capture the style of the artistic image, a measure of the difference between the style features of the generated image and the style image is calculated. This difference is quantified using the **Gram matrix**, which represents the correlations between feature maps in the neural network. The **style loss** measures how well the generated image matches the style of the style image.
3. **GramMatrix**:
   The Gram matrix is a method to capture the **correlation** between different channels (feature maps) of the image. It is computed by multiplying the feature map by its transpose. This matrix captures texture patterns, which are crucial for replicating an image's artistic style.

Loss Function:

The optimization in NST aims to minimize the **total loss**, which is a combination of the content and style losses. This is done iteratively by adjusting the pixels of the generated image.

- **Content Loss** is calculated as the difference between the content representation of the content image and the generated image.
- **Style Loss** is calculated as the difference between the style representation of the style image and the generated image, using the Gram matrix.
- The total loss function combines these two losses with respective weights, allowing the optimization process to balance content retention and style reproduction.

Optimization:

The optimization process uses **gradient descent** to minimize the total loss. By iteratively adjusting the pixels of the generated image, the model gradually creates an image that has the content of the original image and the style of the style image. The process typically uses an optimization algorithm such as **Adam** or **L-BFGS**.

Applications of Neural Style Transfer:

1. **ArtisticRendering**:
   NST can be used to generate artistic representations of photographs by applying the style of famous artists (e.g., Van Gogh, Picasso) to a regular image. This has widespread applications in digital art and design.
2. **PhotoEnhancement**:
   NST allows for the transformation of everyday photographs into visually engaging works of art by transferring the style of selected artworks onto the images.
3. **CreativeIndustries**:
   Artists and designers leverage NST to explore new creative possibilities by combining different art styles with real-world photographs, offering novel ways to express creativity.

Source Code –

```
# Import Required Libraries

import tensorflow as tf

import numpy as np

 import matplotlib.pyplot as plt f

rom tensorflow.keras.preprocessing

import image from tensorflow.keras.applications import vgg19

# Load and Preprocess Image

def load_and_process_img(img_path,target_size=(128, 128)):

img = image.load_img(img_path, target_size=target_size)

img = image.img_to_array(img)

img = np.expand_dims(img, axis=0)

img = vgg19.preprocess_input(img)

return img

def deprocess_img(img):

 img = img.squeeze() img = img + [103.939, 116.779, 123.68] # Convert back to BGR

 img = np.clip(img, 0, 255).astype('uint8')

return img

# Load the Content and Style Images

 content_path = r"C:\Users\saira\Downloads\Content.jpg"

style_path = r"C:\Users\saira\Downloads\style.jpg"

content_img = load_and_process_img(content_path)

style_img = load_and_process_img(style_path)

# Load the VGG19 Model

model = vgg19.VGG19(weights='imagenet', include_top=False)
```

```
# Define the layers we will use for content and style
 content_layer = 'block5_conv2' # Content layer
style_layers = ['block1_conv1', 'block2_conv1', 'block3_conv1', 'block4_conv1', 'block5_conv1']
# Define a Function to Get Feature Representations
def get_feature_representations(model, content_img, style_img):
content_output = model.get_layer(content_layer).output
style_outputs = [model.get_layer(layer).output for layer in style_layers]
model = tf.keras.models.Model(inputs=model.input, outputs=[content_output] + style_outputs)
content_features = model(content_img)
style_features = model(style_img)
 content_feature = content_features[0]
style_features = style_features[1:]
return content_feature, style_features
# Compute the Losses (Content, Style, and Total Variation Loss)
def compute_content_loss(content, generated):
return tf.reduce_mean(tf.square(content - generated))
def compute_style_loss(style, generated):
gram_style = gram_matrix(style)
gram_generated = gram_matrix(generated)
return tf.reduce_mean(tf.square(gram_style - gram_generated))
 def gram_matrix(x):
 channels = int(x.shape[-1])
a = tf.reshape(x, [-1, channels])
gram = tf.matmul(a, a, transpose_a=True)
return gram
```

```
def compute_total_variation_loss(generated):

return tf.reduce_mean(tf.image.total_variation(generated))

# Compute the Gradients and Perform Optimization

def compute_loss(content_feature, style_features, generated_img):

generated_content_feature, generated_style_features = get_feature_representations(model,
generated_img, generated

content_loss = compute_content_loss(content_feature, generated_content_feature)

 style_loss = sum(compute_style_loss(style, generated) for style, generated in zip(style_features,
generated_style

total_variation_loss = compute_total_variation_loss(generated_img)

total_loss = content_loss + 0.025 * style_loss + 1.0 * total_variation_loss

 return total_loss

 def compute_gradients(content_feature, style_features, generated_img):

with tf.GradientTape() as

tape: tape.watch(generated_img)

loss = compute_loss(content_feature, style_features, generated_img)

 grads = tape.gradient(loss, generated_img)

return grads

# Initialize and Optimize the Generated Image

generated_img = tf.Variable(content_img, dtype=tf.float32)

content_feature, style_features = get_feature_representations(model, content_img, style_img)

optimizer = tf.optimizers.Adam(learning_rate=0.01)

iterations = 200

for i in range(iterations):

grads = compute_gradients(content_feature, style_features, generated_img)
optimizer.apply_gradients([(grads, generated_img)])

 if i % 100 == 0:
```

```
print(f"Iteration {i}")
img = deprocess_img(generated_img.numpy())
plt.imshow(img)
 plt.show()
# Display the Final Image
final_img = deprocess_img(generated_img.numpy())
 plt.imshow(final_img)
 plt.show()
```

Conclusion: Neural Style Transfer demonstrates the ability of deep learning models to perform creative tasks by extracting content and style features from images and recombining them. This practical not only deepens the understanding of CNNs but also highlights their potential in applications beyond traditional image classification, such as artistic generation and creative design.