CL3

1. 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.

- **Distributed Systems**

A **distributed system** consists of multiple independent computers that appear to the users as a single coherent system. These systems communicate and coordinate their actions by passing messages.

- **Remote Procedure Call (RPC)**

RPC is a protocol that allows a program to **invoke a procedure (function) on another machine** as if it were local. The details of network communication are abstracted from the user.

Key Components of RPC:

- **Client stub**: Interface for client-side communication.
- **Server stub (skeleton)**: Receives the request and invokes the actual procedure.
- **RPC runtime**: Handles the underlying communication (marshalling, transmission, etc.).

Steps in RPC Execution:

1. Client calls a local stub function.
2. Stub packs the arguments (marshalling).
3. Message is sent to server over the network.
4. Server stub receives, unpacks arguments, and calls actual procedure.
5. Procedure executes, result is returned to server stub.
6. Server stub sends response to client.
7. Client stub receives and unpacks the result.

- **Remote Function**

  A **remote function** is a function that is **executed on a different computer (server)** rather than on the computer where it is called (client). It behaves **just like a local function call** from the client's point of view, but the actual work is done on a **remote machine**.

Application

Microservices: User Authentication,Payement service

Cloud service

Banking System

CODE EXPLAINATION

Let's break down both the **client** and **server** parts of your code and understand how they work together using **XML-RPC**, which is a simple Remote Procedure Call (RPC) protocol in Python.

🧠 **Basic Overview**

- **Client**: Sends a number to the server and requests the factorial.
- **Server**: Receives the number, computes the factorial, and sends the result back.

Both communicate via **HTTP over port 8000** using **XML-RPC**.

✅ **SERVER CODE (Explanation)**

```
from xmlrpc.server import SimpleXMLRPCServer
```

- Imports the module needed to create an XML-RPC server.

```
def factorial(n):
    if n < 0:
        return "Error: Negative numbers not allowed"
    result = 1
    for i in range(2, n + 1):
        result *= i
    return str(result)  # Return as string to avoid integer limits
```

- This is the **remote function**.
- It checks for negative input and returns an error string.
- Otherwise, it calculates factorial using a loop.
- Returns the result as a **string** to avoid problems with very large integers.

```
server = SimpleXMLRPCServer(("localhost", 8000), allow_none=True)
```

- Starts an XML-RPC server on the **localhost (your own machine)** at port 8000.
- allow_none=True allows None values in communication (though not needed here).

```
print("RPC Server started on port 8000...")
server.register_function(factorial, "factorial")
```

- Registers the factorial function as a **remote function** with the name "factorial".
- Now the server can receive remote calls asking it to run factorial(n).

```
server.serve_forever()
```

- Starts the server and waits for incoming requests **forever**.

✅ **CLIENT CODE (Explanation)**

import xmlrpc.client

- Imports the module to make **RPC requests** from the client side.

---

proxy = xmlrpc.client.ServerProxy("http://localhost:8000/")

- Connects to the XML-RPC server at the specified address (localhost:8000).
- proxy acts like a **remote object** — any method you call on it is sent to the server.

---

num = int(input("Enter an integer to compute factorial: "))

- Takes an integer input from the user.

---

result = proxy.factorial(num)

- Calls the **remote function** factorial() on the server, passing num as argument.
- This call goes through the network to the server.

---

print(f"Factorial of {num} is:\n{result}")

- Prints the result received from the server.

🔁 **How They Work Together**

User → Client → RPC Call → Server → Factorial Computation → Return Result → Client → Display

- The client feels like it's calling a normal function.
- The server does the work and sends back the result.

Difference between RPC and RMI

✅ **Difference Between RPC and RMI**

You likely meant **RMI** (Remote Method Invocation) instead of **RML**, which is a different concept unrelated to distributed computing.

So here's a clear comparison between **RPC (Remote Procedure Call)** and **RMI (Remote Method Invocation)**:

Feature	**RPC (Remote Procedure Call)**	**RMI (Remote Method Invocation)**
Full Form	Remote Procedure Call	Remote Method Invocation

Feature	RPC (Remote Procedure Call)	RMI (Remote Method Invocation)
Language Support	Language-agnostic (C, Python, Java, etc.)	Java-specific (used only in Java)
Function Type	Calls **procedures/functions** (like factorial(x))	Calls **methods** on **remote objects**
Protocol Format	Uses protocols like **XML-RPC**, **JSON-RPC**, **gRPC**	Uses **Java RMI Protocol (JRMP)**
Data Representation	Simple data types (int, string, list, etc.)	Can use **complex Java objects**, which are serialized
Platform Dependency	Platform-independent	**Java-only**, not cross-platform
Object-Oriented Support	Not inherently object-oriented	Fully supports OOP (calls methods on objects)
Use Case Example	Python client calling a C server to compute a value	Java client invoking a method on a Java object on the server
Example Tool	xmlrpc.client in Python	java.rmi.registry, UnicastRemoteObject

---

🔍 **Summary:**

- **RPC** is **simpler and more general**, good for calling remote **functions** in different languages.
- **RMI** is **Java-specific** and more **object-oriented**, allowing you to invoke **methods on remote Java objects**.

Mechanism	Protocol(s) Used	Description
RPC (in general)	TCP/IP, HTTP, gRPC, SOAP, XML-RPC, JSON-RPC	- Traditional RPC uses **TCP/IP** for sending requests/responses.- Modern implementations (like gRPC) use **HTTP/2** and **Protocol Buffers**.- In Python,

Mechanism	Protocol(s) Used	Description
		xmlrpc.client uses **HTTP + XML** to encode data.
Pyro4 (Python RPC)	TCP/IP (custom Pyro protocol)	Pyro4 uses its own **custom protocol over TCP/IP** for fast communication.It serializes data using **pickle** or **serpent** formats.
RMI (Java)	JRMP (Java Remote Method Protocol)	Java RMI uses **JRMP**, a Java-specific protocol built over TCP/IP.It handles Java object serialization and method invocation over the network.

2. 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.

✅ **What It Means**

You're asked to **create a distributed application** using **Java RMI (Remote Method Invocation)** where:

- A **client** sends **two strings** to a **server**.
- The **server combines (concatenates)** the two strings and sends the **result back** to the client.

🧠 **Conceptual Breakdown**

Component	Explanation
Distributed Application	A program that runs across **two or more computers** connected via network.
RMI (Remote Method Invocation)	A Java mechanism that allows one Java object (client) to invoke methods on another object (server) located remotely.
Client	The computer/user that sends two strings (e.g., "Hello" and "World").

Component	Explanation
Server	The computer that receives the strings, performs concatenation ("HelloWorld"), and sends the result back.

📌 **Functional Flow**

1. **Client** enters two strings (e.g., "Data", "Science").
2. The strings are sent to the **server** over the network.
3. The **server's remote method** performs:
4. return str1 + str2;
5. Server returns the result back to the **client**.
6. Client displays:
7. Concatenated String: DataScience

🧩 **Java RMI Application Components**

To implement this in Java RMI, you need 4 main components:

1. **Remote Interface**
   - Declares the remote method (e.g., String concatenate(String a, String b)).
2. **Server Implementation**
   - Implements the remote method (i.e., the logic to concatenate strings).
3. **Server Program**
   - Registers the remote object to the RMI registry so clients can access it.
4. **Client Program**
   - Connects to the server via RMI registry and invokes the remote method.

🧪 **Sample Use-Case**

- A Java app that allows users to send words to a language-processing server and combine them for sentence generation.
- For example:
  - Input: "Open", "AI"
  - Output: "OpenAI"

OR

Absolutely! Here's the **theory explanation** for your Python-based distributed application using **Pyro4**:

📝 **Theory for Distributed Application using Pyro4**

🔷 **Objective**

To design a distributed application in **Python** using the **Pyro4** library where:

- A **client** submits two strings to a **server**.
- The **server** returns the **concatenation** of the given strings.

🔷 What is Pyro4?

Pyro4 (Python Remote Objects) is a library that enables you to:

- Build **distributed applications** in Python.
- **Call remote methods** just like local ones.
- Use **remote objects** running on other machines or processes.

Pyro4 is Python's version of **Remote Method Invocation (RMI)**.

🔷 Concepts Used

Concept	Description
Remote Object	A Python object whose methods can be called remotely by a client.
Proxy Object	A local reference to the remote object on the client side.
URI (Uniform Resource Identifier)	A unique address used to locate the remote object in the network.
Pyro4.Daemon	Creates a server that listens for client requests.
@Pyro4.expose	Makes a class or method accessible to clients over the network.
Threading	Allows the server to run in the background without blocking the main program.

🔷 System Components

1. **Server Program**:
   - Defines and exposes a remote class StringConcatenator with a method concatenate(str1, str2).
   - Starts a Pyro daemon to **register the object** and **wait for client requests**.
2. **Client Program**:
   - Connects to the server using the URI of the remote object.
   - Sends two strings to the server using the concatenate() method.
   - Displays the concatenated result returned by the server.

🔷 Working of the Application

1. **Server Side**:
    - The server creates an instance of Pyro4.Daemon().
    - The StringConcatenator object is **registered** with the daemon.
    - The daemon prints the **URI**, which acts like the "address" for the remote object.
    - The server waits for client requests via requestLoop().
2. **Client Side**:
    - The client connects to the remote object using the provided **URI**.
    - Sends two string values to the server.
    - Calls the remote concatenate() method.
    - Receives the result and prints it.

🔷 **Advantages of Using Pyro4**

- Pure Python implementation: No need for other languages.
- Very similar to **local method calls**.
- Easy to build **networked and modular applications**.
- Simple syntax and architecture.

🔷 **Example Use Case**

Imagine you're building a **microservice-based system**, where:

- One service handles user input.
- Another backend service performs text processing (e.g., merging names or messages).

Pyro4 allows you to distribute these services across different systems and keep them decoupled.

CODE EXPLAINATION

You're implementing a **distributed application using Pyro4**, which is a Python library for Remote Method Invocation—very similar in concept to Java RMI. Here's a detailed explanation of both the **server** and **client** code.

🔧 **Concept Overview**

- **Goal**: A client sends two strings to the server. The server concatenates them and returns the result.
- **Library Used**: Pyro4 (Python Remote Objects) — It allows remote method calls between Python programs, similar to Java RMI.

🖥️ **SERVER EXPLANATION**

```
import Pyro4
import threading
```

- **Pyro4** is the library that allows defining and using remote objects.
- **threading** allows us to run the server in the background.

Remote Class Definition

```
@Pyro4.expose
class StringConcatenator:
    def concatenate(self, str1, str2):
        print(f"Received: {str1}, {str2}")
        return str1 + str2
```

- This is the **remote object** whose method concatenate() will be called by the client.
- @Pyro4.expose allows this class and its method to be used remotely.
- It simply prints the received strings and returns the **concatenation**.

Starting the Server

```
def start_server():
    daemon = Pyro4.Daemon(host="localhost")  # Start server daemon
    uri = daemon.register(StringConcatenator)  # Register remote object
    print("Server running. URI:", uri)
    daemon.requestLoop()  # Keep server listening for requests
```

- **Pyro4.Daemon()** creates a communication endpoint on the host machine.
- **register()** gives the object a URI so clients can find and connect to it.
- **requestLoop()** starts an infinite loop that waits for remote calls.

Running the Server in Background

```
thread = threading.Thread(target=start_server)
thread.daemon = True
thread.start()
```

- The server is started in a **background thread**, allowing you to interact with the rest of your notebook or program without blocking it.

CLIENT EXPLANATION

```
import Pyro4
```

- Import the same Pyro4 library to access the remote server object.

Connecting to Remote Object

```
uri = "PYRO:obj_0a8271e3b1534875a907f2d223d14a08@localhost:57684"  # Use actual URI
proxy = Pyro4.Proxy(uri)
```

- Use the URI printed by the server to connect.
- proxy is a **remote reference** to the StringConcatenator object on the server.

Sending Data and Receiving Result

```
str1 = input("Enter first string: ")
str2 = input("Enter second string: ")
result = proxy.concatenate(str1, str2)
print("Concatenated result:", result)
```

- Takes input strings from the user.
- Calls the concatenate() method on the server.
- Displays the result returned by the server.

🔁 **Flow Summary**

Client: Enter "Hello", "World"
↓
Client → Pyro Proxy → Server
↓
Server runs concatenate("Hello", "World")
↓
Returns "HelloWorld" to client
↓
Client prints result

3. 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.

1. Union of Fuzzy Sets

The **union** of two fuzzy sets AAA and BBB, denoted as A∪BA \cup BA∪B, combines the membership degrees of elements in either set.

- **Formula**:

$$\mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

where μA(x) and μB(x) are the membership degrees of x in fuzzy sets A and B, respectively.

Explanation:
The membership degree of an element x in the union of fuzzy sets is the maximum of the membership degrees of x in both sets. This operation reflects that an element belongs to the union if it belongs to either set.

2. Intersection of Fuzzy Sets

The **intersection** of two fuzzy sets AAA and BBB, denoted as A∩BA \cap BA∩B, combines the membership degrees by taking the **minimum** value between the two sets.

- **Formula**:

$$\mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

where μA(x) and μB(x) are the membership degrees of x in fuzzy sets A and B, respectively.

Explanation:

The membership degree of an element xxx in the intersection of fuzzy sets is the minimum of the membership degrees of xxx in both sets. This operation reflects that an element belongs to the intersection if it belongs to both sets to some degree.

3. Complement of a Fuzzy Set

The **complement** of a fuzzy set A, denoted as A', reflects the idea of **non-membership** in the set.

- **Formula**:

$$\mu_{A'}(x) = 1 - \mu_A(x)$$

where μA(x) is the membership degree of x in fuzzy set A.

Explanation:

The membership degree of an element xxx in the complement of fuzzy set A is the difference between 1 and the membership degree of x in A. If an element is highly in the set, it will have a low membership in its complement.

4. Difference of Fuzzy Sets

The **difference** between two fuzzy sets AAA and BBB, denoted as A−BA - BA−B, is a measure of how much AAA is distinct from BBB.

- **Formula**:

$$\mu_{A-B}(x) = \max(0, \mu_A(x) - \mu_B(x))$$

where μA(x) and μB(x) are the membership degrees of x in fuzzy sets A and B, respectively.

Explanation:

The membership degree of an element x in the difference of fuzzy sets A and B is the difference between the membership degree of x in A and its membership degree in B, clipped at 0. This reflects that the element x is more in A than in B.

Fuzzy Relations: Cartesian Product of Fuzzy Sets

A **fuzzy relation** is a mapping between elements of two fuzzy sets, where each element in the Cartesian product of the sets has a membership degree in the relation.

- If AAA and BBB are two fuzzy sets, their **Cartesian product** R=A×B is a **fuzzy relation** that relates each pair of elements (x,y)from A and B.
- **Membership Function**:

$$\mu_R(x, y) = \min(\mu_A(x), \mu_B(y))$$

- This means that the membership degree of the pair (x,y) in the fuzzy relation is the **minimum** of the membership degrees of x in A and y in B.

Max-Min Composition of Fuzzy Relations

The **max-min composition** of two fuzzy relations is a way of combining them to form a new fuzzy relation. This operation is often used in fuzzy logic systems for **transitive closure** or **fuzzy inference systems**.

- If RRR is a fuzzy relation from A to B, and S is a fuzzy relation from B to C, then the **composition** of R and S, denoted as R∘S, is a fuzzy relation from A to C.
- **Max-Min Composition Formula**:

$$\mu_{R \circ S}(x, z) = \max_{y \in B} [\min(\mu_R(x, y), \mu_S(y, z))]$$

 where:

- μR(x,y) is the membership degree of the pair (x,y) in relation R,
- μS(y,z) is the membership degree of the pair (y,z) in relation S,
- The max-min operation is used to aggregate the results over all possible values of y from B.

 Explanation:

The max-min composition computes the membership degree for each pair (x,z) in the new relation by combining the relations R and S through a **minimum** operation for each possible intermediary element y, and then selecting the **maximum** of all those values.

CODE EXPLAINATION

Explanation of the Code

This Python code demonstrates various operations on **fuzzy sets** and **fuzzy relations**. It performs set operations such as **union**, **intersection**, **complement**, and **difference** on fuzzy sets. It also performs **Cartesian product** and **max-min composition** on fuzzy relations. Let's break down the functionality and explain the operations:

---

1. Union Operation (fuzzy_union)

The union of two fuzzy sets AA and BB is computed by comparing the corresponding membership degrees of elements from both sets and taking the maximum value for each element.

- **Input**: Two dictionaries representing fuzzy sets AA and BB.
- **Output**: A dictionary where each element is the maximum of the corresponding elements in sets AA and BB.

```
def fuzzy_union(a,b):
    union ={}
    for key in a.keys():
        union[key] = max(a[key],b[key])
    return union
```

Explanation:

- For each key in the dictionary aa, the function takes the maximum of the corresponding values in aa and bb, and stores the result in the dictionary union.
- The max() function is used because the union of fuzzy sets reflects the highest degree of membership.

Example Output:

```
Union :
 {'x1': 0.4, 'x2': 0.7, 'x3': 0.5, 'x4': 0.5, 'x5': 0.9}
```

---

2. Intersection Operation (fuzzy_intersection)

The intersection of two fuzzy sets AA and BB is computed by comparing the corresponding membership degrees and taking the minimum value for each element.

- **Input**: Two dictionaries representing fuzzy sets AA and BB.
- **Output**: A dictionary where each element is the minimum of the corresponding elements in sets AA and BB.

```
def fuzzy_intersection(a,b):
    intersection = {}
    for key in a.keys():
        intersection[key] = min(a[key],b[key])
    return intersection
```

Explanation:

- The min() function is used because the intersection of fuzzy sets reflects the lowest degree of membership.
- For each key in the dictionary aa, the minimum of the corresponding values in aa and bb is stored in the dictionary intersection.

Example Output:

```
Intersection:
 {'x1': 0.2, 'x2': 0.6, 'x3': 0.3, 'x4': 0.1, 'x5': 0.2}
```

---

3. Complement Operation (complement)

The complement of a fuzzy set AA is computed by subtracting the membership degree of each element from 1.

- **Input**: A dictionary representing fuzzy set AA.
- **Output**: A dictionary where each element is 1−μA(x)1 - \mu_A(x).

```
def complement(a):
    comp_a ={}
    for key in a.keys():
        comp_a[key] = 1 - a[key]
    return comp_a
```

Explanation:

- For each element in set AA, the function computes the complement by subtracting its membership degree from 1.
- This operation is used to represent non-membership of elements in fuzzy logic.

Example Output:

```
Complement :
 {'x1': 0.8, 'x2': 0.4, 'x3': 0.7, 'x4': 0.5, 'x5': 0.1}
```

---

4. Difference Operation (diff)

The difference between two fuzzy sets AA and BB is computed by subtracting the membership degree of corresponding elements in BB from 1 and taking the minimum value between this and the membership degree in AA.

- **Input**: Two dictionaries representing fuzzy sets AA and BB.
- **Output**: A dictionary where each element is min⁡(μA(x),1−μB(x))\min(\mu_A(x), 1 - \mu_B(x)).

```
def diff(a,b):
    diff ={}
    for key in a.keys():
        diff[key] = min(a[key], 1 - b[key])
    return diff
```

Explanation:

- For each key in the dictionary aa, the function computes the difference by taking the minimum between the membership degree of xx in AA and 1−μB(x)1 - \mu_B(x) (which represents the complement of xx in BB).
- This operation reflects how much an element in AA is distinct from the element in BB.

Example Output:

```
Difference :
 {'x1': 0.2, 'x2': 0.6, 'x3': 0.3, 'x4': 0.4, 'x5': 0.9}
```

---

5. Cartesian Product (cart_product)

The **Cartesian product** of two fuzzy sets AA and BB is a fuzzy relation, where each pair of elements (x,y)(x, y) from A×BA \times B has a membership degree computed as the minimum of their corresponding membership degrees from AA and BB.

- **Input**: Two dictionaries representing fuzzy sets AA and BB.
- **Output**: A dictionary representing the fuzzy relation R=A×BR = A \times B, where the key is the tuple (x,y)(x, y) and the value is the membership degree.

```
def cart_product(a,b):
  prod = {}
  for a_key in a.keys():
    for b_key in b.keys():
      prod[(a_key,b_key)] = min(a[a_key],b[b_key])
  return prod
```

Explanation:

- For each pair of elements from AA and BB, the membership degree in the product relation is the minimum of their membership degrees in AA and BB.

Example Output:

```
Product :
 {('x1', 'y1'): 0.2, ('x1', 'y2'): 0.4, ('x2', 'y1'): 0.6, ('x2', 'y2'): 0.6}
```

---

6. Max-Min Composition of Fuzzy Relations (max_min_composition)

The **max-min composition** is a fuzzy operation that combines two fuzzy relations RR and SS using the **min** and **max** operators. The goal is to find the "path" from an element in the domain of RR to an element in the codomain of SS by aggregating the minimum values from each intermediate relation.

- **Input**: Two fuzzy relations RR (from X→YX \to Y) and SS (from Y→ZY \to Z).
- **Output**: A dictionary representing the composition of RR and SS, from X→ZX \to Z, computed using the **max-min** composition.

```
def max_min_composition(R, S):
  result = {}
  for x, y1 in R:
    for y2, z in S:
      if y1 == y2:
        pair = (x, z)
        value = min(R[(x, y1)], S[(y2, z)])
        result[pair] = max(result.get(pair, 0), value)
  return result
```

Explanation:

- The function iterates through all pairs in RR and SS, matching intermediate elements y1y1 and y2y2.
- The membership degree for the pair (x,z)(x, z) is calculated as the minimum of the membership degrees from RR and SS.

- The **max** operator is used to combine the values for the same (x,z)(x, z) pair across different paths.
  Example Output:
  Min Max Composition :
  {('x1', 'z1'): 0.2, ('x1', 'z2'): 0.4, ('x2', 'z1'): 0.6, ('x2', 'z2'): 0.8}

4. Write code to simulate requests coming from clients and distribute them among the servers using the load balancing algorithms.

What is Load Balancing?
Load balancing is a method to distribute incoming network traffic or computing tasks across multiple servers, ensuring **no single server is overwhelmed**, and resources are utilized **efficiently**.

📦 **Common Load Balancing Algorithms:**

1. **Round Robin**:
   - Assigns requests to servers in a circular order.
   - Simple and doesn't consider server load.
2. **Least Connections**:
   - Assigns the request to the server with the **fewest active connections**.
3. **Random Selection**:
   - Randomly chooses a server to handle the request.

✅ **What is a Load Balancer?**
A **Load Balancer** is a system (hardware or software) that **distributes incoming network traffic or client requests across multiple servers** to ensure:

- **Efficient resource utilization**
- **High availability**
- **Fault tolerance**
- **Better response time**

🧠 **Why is Load Balancing Needed?**
When many clients send requests (like on a website or app), relying on a single server can cause:

- Overload (slow responses)
- Crashes or downtime
- Poor user experience

A load balancer helps avoid this by **spreading the load** evenly across multiple servers.

🛠️ **How Does It Work?**

1. A client sends a request to a service.

2. The load balancer **sits in between** the client and the backend servers.
3. It forwards the request to one of the servers based on a **load balancing algorithm** (e.g., round robin, least connections, random).
4. The selected server processes the request and sends back the result.

📊 **Common Load Balancing Algorithms**

Algorithm	Description
Round Robin	Assigns requests to servers one by one in order.
Least Connections	Sends request to the server with the fewest active connections.
Random	Chooses a random server for each request.
Weighted Round Robin	Some servers can handle more load and are assigned more requests.

Here are the steps involved in the **three common load balancing algorithms**: Round Robin, Least Connections, and Random — as implemented in your Python code.

⚙️ **1. Round Robin Algorithm**

🔁 **Purpose**: Distributes requests in a fixed circular order.

Steps:

1. Start with an index (round_robin_index = 0).
2. When a client request arrives:
   - Assign it to the server at the current index.
   - Start a new thread to simulate request handling.
3. Increment the index:
   round_robin_index = (round_robin_index + 1) % number_of_servers
4. Repeat for the next request.

📝 Example:

If there are 3 servers and 5 clients, assignment goes like:

Client1 → Server1

Client2 → Server2

Client3 → Server3

Client4 → Server1 (starts again)

Client5 → Server2

⚙️ **2. Least Connections Algorithm**

📉 **Purpose**: Sends request to the server that's currently the least busy.

Steps:

1. When a client request arrives:
    - Check the active_connections count for all servers.
    - Select the server with the **minimum** active connections.
    - Assign the request to this server and start a new thread.
2. Repeat for the next request.

📝 Example:

If servers have active connections like:

- Server1: 3
- Server2: 1
- Server3: 2

→ Client will be assigned to **Server2**

⚙️ **3. Random Algorithm**

🎲 **Purpose**: Selects a server at random for each request.

Steps:

1. When a client request arrives:
    - Use random.choice(servers) to pick a server.
    - Assign the request to this randomly selected server.
    - Start a new thread to simulate request handling.
2. Repeat for the next request.

📝 Example:

Clients might be assigned in this unpredictable order:

Client1 → Server2

Client2 → Server1

Client3 → Server3

Client4 → Server1

...

CODE EXPLAINATION

This Python code is a **simulation of client requests** being distributed to servers using different **load balancing algorithms**. It uses **threading** to mimic real-time request handling by multiple servers.

Let's break it down section by section:

🔧 **1. Server Class**

```
class Server:
  def __init__(self, name):
    self.name = name
    self.active_connections = 0
```

- Each Server has a name and a count of active_connections.

  🔄 **handle_request() Method:**

  ```
  def handle_request(self):
      self.active_connections += 1
      print(f"Server {self.name} is handling a request. Active connections: {self.active_connections}")
      time.sleep(2)
      self.active_connections -= 1
      print(f"Server {self.name} has finished handling a request. Active connections: {self.active_connections}")
  ```

- Simulates a server handling a client request.
- Increases active_connections, waits 2 seconds (simulating work), then decreases it.
- time.sleep(2) mimics request processing time.

⚖️ **2. LoadBalancer Class**

```
class LoadBalancer:
    def __init__(self, servers, algorithm="round_robin"):
        self.servers = servers
        self.algorithm = algorithm
        self.round_robin_index = 0
```

- Accepts a list of servers and a chosen algorithm.
- round_robin_index helps keep track of which server to assign next in **round robin**.

  🧠 **distribute_request() Method:**

  ```
  def distribute_request(self, client_id):
      if self.algorithm == "round_robin":
          self.round_robin(client_id)
      elif self.algorithm == "least_connections":
          self.least_connections(client_id)
      elif self.algorithm == "random":
          self.random(client_id)
  ```

- Chooses the load balancing algorithm and calls the appropriate method.

2.1. 🪆 **round_robin():**

```
def round_robin(self, client_id):
    server = self.servers[self.round_robin_index]
    ...
    self.round_robin_index = (self.round_robin_index + 1) % len(self.servers)
    threading.Thread(target=server.handle_request).start()
```

- Assigns requests in a rotating (round-robin) fashion.
- Uses threading.Thread to simulate asynchronous server handling.

2.2. 📉 least_connections():

```
def least_connections(self, client_id):
    server = min(self.servers, key=lambda s: s.active_connections)
    ...
```

- Finds the server with the **fewest active connections** and assigns the request.

2.3. 🎲 random():

```
def random(self, client_id):
    server = random.choice(self.servers)
    ...
```

- Randomly picks a server from the list.

👥 3. simulate_clients() Function

```
def simulate_clients(load_balancer, num_clients):
    for client_id in range(1, num_clients + 1):
        load_balancer.distribute_request(client_id)
        time.sleep(random.uniform(0.5, 1.5))
```

- Simulates multiple clients sending requests.
- Each client is assigned to a server using the selected algorithm.
- sleep() adds a random delay between client arrivals.

🟢 4. Main Execution Block

```
if __name__ == "__main__":
    servers = [Server(name=f"Server-{i}") for i in range(1, 4)]
    lb = LoadBalancer(servers=servers, algorithm="round_robin")
    simulate_clients(lb, num_clients=10)
```

- Initializes 3 servers.
- Creates a LoadBalancer using **round robin**.
- Simulates **10 client requests**.

🧠 Summary

Component	Purpose
Server class	Represents a server that handles client requests.
LoadBalancer	Distributes requests using one of three algorithms.
Threading	Allows simultaneous request handling.
simulate_clients()	Mimics multiple clients sending requests.

5.Optimization of genetic algorithm parameter in hybrid genetic algorithm-neural network modelling: Application to spray drying of coconut milk.

🔷 What is a Genetic Algorithm (GA)?

A **Genetic Algorithm** is a **search and optimization technique** inspired by the process of **natural selection** in biological evolution. It is especially useful for solving complex problems where traditional methods struggle — like feature selection, neural network optimization, or scheduling.

It mimics the behavior of biological evolution using key concepts such as **population**, **genes**, **selection**, **crossover**, and **mutation** to evolve solutions to problems over generations.

🔷 Basic Terminology

Term	Meaning
Chromosome	A single solution (usually encoded as a string or list)
Population	A group of candidate solutions
Gene	A part of the chromosome (e.g., one parameter or feature)
Fitness	A measure of how good a solution is at solving the problem
Generation	One complete cycle of evolution (selection + crossover + mutation)

🔷 Steps in Genetic Algorithm

1. **Initialization**
    - Generate a random **initial population** of candidate solutions (chromosomes).
    - The size of the population is usually fixed.

🧠 *Example:* If we are optimizing 3 parameters, a chromosome might look like [0.5, 0.8, 0.3].

2. **Fitness Evaluation**
    - Each chromosome is evaluated using a **fitness function** that measures how well it solves the problem.

🧠 *Example:* For a neural network, fitness could be inverse of prediction error (like 1 / MSE).

3. **Selection**

- Select the best solutions (parents) from the population based on fitness.
- Methods: **Roulette Wheel**, **Tournament Selection**, etc.

🧠 *Better chromosomes have a higher chance of being selected for the next generation.*

4. **Crossover (Recombination)**
    - Combine pairs of parents to form **new offspring** (children).
    - This allows mixing of good traits.

🧠 *Example:* Parents [0.5, 0.8, 0.3] and [0.2, 0.6, 0.9] can produce [0.5, 0.6, 0.9].

5. **Mutation**
    - Introduce small random changes in offspring to maintain **genetic diversity**.
    - This helps avoid **local optima**.

🧠 *Example:* [0.5, 0.6, 0.9] → [0.5, 0.65, 0.9] (random tweak to second gene).

6. **Replacement**
    - The new generation (offspring) replaces some or all of the old population.
    - The process repeats over several **generations**.
7. **Termination**
    - Stop when a stopping condition is met:
        - A solution with **acceptable fitness** is found.
        - A **maximum number of generations** is reached.
        - Fitness improvement has **stalled** over generations.

🔚 **Summary Table**

Step	Purpose
Initialization	Create a random population
Fitness Evaluation	Measure solution quality
Selection	Choose best solutions as parents
Crossover	Mix parent genes to create children

Step	Purpose
Mutation	Introduce randomness
Replacement	Form new population
Termination	End process when criteria are met

CODE EXPLAINATION

This code demonstrates a **hybrid Genetic Algorithm–Neural Network model** for predicting the **moisture content** in the **spray drying process of coconut milk**, based on simulated data. Let's break it down **line by line and section by section** for clarity:

🔹 1. Import Libraries

```
import numpy as np
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
import random
```

- numpy: For numerical operations.
- MLPRegressor: A neural network regressor from sklearn.
- mean_squared_error: Used to evaluate prediction error.
- train_test_split: To divide the dataset into training and testing sets.
- random: Python's random module for crossover/mutation logic.

🔹 2. Simulate Dataset

```
np.random.seed(42)
X = np.random.uniform(low=150, high=200, size=(100, 3))  # features
y = 0.3*X[:,0] - 0.2*X[:,1] + 0.1*X[:,2] + np.random.normal(0, 2, 100)  # target
```

- Simulate 100 data samples with 3 features (InletTemp, FeedRate, AtomSpeed).
- Target y (Moisture Content) is generated using a known formula + noise.

🔹 3. Split Data into Train/Test

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

- Splits the dataset into **80% training** and **20% testing**.

🔹 4. Neural Network with Custom Weights

```
def create_nn(weights):
    model = MLPRegressor(hidden_layer_sizes=(5,), max_iter=1, warm_start=True)
    model.fit(X_train, y_train)
```

- A simple neural network with **1 hidden layer of 5 neurons**.
- max_iter=1 and warm_start=True allow setting weights manually.
- It is fit once so internal structures are initialized.

```
    weights = np.array(weights)
    i = 0
    for layer_weights in model.coefs_:
        shape = layer_weights.shape
        model.coefs_[i] = weights[:np.prod(shape)].reshape(shape)
        weights = weights[np.prod(shape):]
        i += 1
```

- Injects the weights from the **genetic algorithm** into the model manually.
- model.coefs_ gives access to the network's weight matrices.

🔹 5. Fitness Function

```
def fitness_function(weights):
    model = create_nn(weights)
    preds = model.predict(X_train)
    return mean_squared_error(y_train, preds)
```

- Measures how good a solution (set of weights) is by computing the **MSE** (lower is better).

🔹 6. Genetic Algorithm Components

▪ Crossover

```
def crossover(p1, p2):
    point = random.randint(0, len(p1)-1)
    return p1[:point] + p2[point:]
```

- Single-point crossover: combines part of p1 and p2 to create a child.

▪ Mutation

```
def mutate(ind, rate=0.1):
    return [w + np.random.randn()*rate if random.random() < 0.1 else w for w in ind]
```

- Slightly changes genes with 10% probability to maintain diversity.

🔹 7. Initialize Population

```
n_weights = (3*5) + (5*1)  # assuming 3 input, 5 hidden, 1 output
pop = [np.random.uniform(-1, 1, n_weights).tolist() for _ in range(10)]
```

- Total weights:
  - 3 inputs × 5 hidden = 15
  - 5 hidden × 1 output = 5
  - Total = 20
- Create 10 individuals with random weights in range [-1, 1].

🔹 8. Run Genetic Algorithm

```
for generation in range(10):
    pop = sorted(pop, key=fitness_function)
    new_pop = pop[:2]  # elitism
```

- Run for 10 generations.

- Sort the population based on fitness (lowest MSE).
- Elitism: directly carry over top 2 individuals.

```
while len(new_pop) < len(pop):
    p1, p2 = random.sample(pop[:5], 2)
    child = mutate(crossover(p1, p2))
    new_pop.append(child)
pop = new_pop
print(f"Gen {generation+1}, MSE: {fitness_function(pop[0]):.4f}")
```

- Select parents randomly from top 5.
- Generate children using crossover and mutation.
- Add them to new population until it reaches original size.
- Print the best fitness (MSE) for each generation.

🔹 9. Final Evaluation

```
best_weights = pop[0]
final_model = create_nn(best_weights)
test_preds = final_model.predict(X_test)
print("Test MSE:", mean_squared_error(y_test, test_preds))
```

- After evolution, best weights are selected.
- Neural network is built using them and evaluated on **unseen test data**.

🔚 Summary

Concept	Purpose
Neural Network	Predict moisture content
Genetic Algorithm	Optimize the weights of the NN
Crossover & Mutation	Explore better solutions
Fitness	Measures how well weights work (via MSE)

Concept	Purpose
Evolution	Improves model weights over generations

6.Implementation of Clonal selection algorithm using Python.

📌 What is the Clonal Selection Algorithm (CSA)?

The **Clonal Selection Algorithm** is a bio-inspired optimization algorithm based on how the human **immune system** defends the body. It mimics the **clonal selection principle** — a theory from immunology that explains how B-cells (a type of white blood cell) clone themselves to fight pathogens (like viruses or bacteria).

CSA is commonly used for solving:

- Optimization problems
- Pattern recognition tasks
- Function approximations

🧠 Key Concepts Behind CSA

Biological Term	Algorithm Equivalent
Antibody	A candidate solution
Antigen	The problem or objective
Affinity	Fitness score (how good a solution is)
Cloning	Copying good solutions
Hypermutation	Slight changes in cloned solutions
Selection	Choosing best solutions for next generation

🔁 Steps of Clonal Selection Algorithm

1. **Initialize** a population of random candidate solutions (antibodies).
2. **Evaluate** the fitness (affinity) of each solution.
3. **Select** the top N best antibodies (those with highest affinity).

4. **Clone** these selected antibodies in proportion to their fitness (more fit → more clones).
5. **Mutate** the clones (hypermutation), especially those with lower affinity.
6. **Evaluate** the new mutated clones.
7. **Replace** the worst antibodies with the best clones.
8. **Repeat** until stopping criteria is met (like number of generations or good enough fitness).

🧬 **Theory Summary**

Step	Purpose
Initialization	Generate random solutions
Affinity Evaluation	Check how well each solution solves the problem
Selection	Keep the top solutions
Cloning	Make copies of good solutions
Mutation	Add randomness to explore new areas
Replacement	Improve the population by removing weak solutions

CODE EXPLAINATION

This code implements a **simplified version of the Clonal Selection Algorithm** using Python. It aims to **find the minimum** of the function $f(x) = x^2$, which has its global minimum at $x = 0$.

1. Imports and Objective Function

```
import numpy as np


# Objective function to minimize
def fitness_function(x):
    return x ** 2
```

- Uses **NumPy** for numerical operations.

- The **fitness function** is x^2, and the algorithm tries to find the value of x that minimizes it (best value = 0).

---

2. Algorithm Parameters

population_size = 10

clone_factor = 3

mutation_rate = 0.1

num_generations = 10

selection_size = 5

- population_size: Number of candidate solutions in each generation.
- clone_factor: How many **clones** are made for each selected "good" solution.
- mutation_rate: Controls the randomness added during mutation.
- num_generations: Number of iterations the algorithm runs.
- selection_size: Number of best solutions chosen for cloning.

---

3. Initialize Population

population = np.random.uniform(-10, 10, population_size)

- Creates 10 random values (antibodies) between -10 and 10.

---

4. Start of Evolution Loop

for generation in range(1, num_generations + 1):

- Runs the entire evolution process for 10 generations.

---

5. Evaluate Fitness

fitness = np.array([fitness_function(x) for x in population])

- Calculates how "good" each solution is using the fitness function.

---

6. Select the Best Antibodies

selected_indices = fitness.argsort()[:selection_size]

selected = population[selected_indices]

- Selects the **top 5 antibodies** with the **lowest fitness values** (closest to 0).

---

7. Cloning

clones = np.repeat(selected, clone_factor)

- Clones each selected antibody **3 times**, creating 5 x 3 = 15 clones.

---

8. Mutation

mutations = clones + np.random.normal(0, mutation_rate, size=clones.shape)

- Slightly modifies (mutates) each clone by adding random noise drawn from a normal distribution.

---

9. Evaluate Mutated Clones

mutated_fitness = np.array([fitness_function(x) for x in mutations])

- Recomputes fitness for all 15 mutated clones.

---

10. Select Best Clones

best_mutated_indices = mutated_fitness.argsort()[:population_size - 1]

new_population = mutations[best_mutated_indices]

- Chooses the best 9 clones to be part of the next population.

---

11. Introduce Diversity

new_random = np.random.uniform(-10, 10, 1)

population = np.concatenate((new_population, new_random))

- Adds 1 completely new random antibody to keep diversity in the population.

---

12. Print Progress

best_fitness = fitness_function(population[np.argmin([fitness_function(x) for x in population])])

print(f"Gen {generation}: Best Fitness = {best_fitness:.4f}")

- Prints the best fitness value for the current generation.

---

13. Final Output

best_solution = population[np.argmin([fitness_function(x) for x in population])]

print("\n=== Final Output ===")

print(f"Best solution: x = {best_solution:.4f}")

print(f"Best fitness: {fitness_function(best_solution):.4f}")

- After all generations, it prints the **best solution** found and its fitness value.

📌 **Summary of Key Concepts Used**

Step	Concept
Initial population	Random candidate solutions (antibodies)
Fitness evaluation	Evaluating how close each value is to optimal
Selection	Choosing top-performing antibodies
Cloning	Copying good antibodies multiple times
Mutation	Small random changes to explore new solutions
Replacement	Keeping the best new solutions
Diversity	Adding a new random antibody

7. To apply the artificial immune pattern recognition to perform a task of structure damage Classification.

Applying **Artificial Immune Systems (AIS)** — specifically **Artificial Immune Pattern Recognition (AIPR)** — to **structure damage classification** involves mimicking the **human immune system's ability** to distinguish between "self" (healthy) and "non-self" (damaged) patterns. This technique is useful in **Structural Health Monitoring (SHM)**, where the goal is

to detect, localize, and classify damages in structures like bridges, buildings, or aircraft components.

📘 Theory Overview

🧠 1. What is Artificial Immune Pattern Recognition (AIPR)?

AIPR is a **bio-inspired algorithm** that simulates how the **biological immune system** recognizes pathogens and foreign substances. The immune system can distinguish between the body's own cells (self) and potentially harmful invaders (non-self).

In pattern recognition tasks (like structure damage classification), this concept is used to distinguish between:

- **Self patterns** = Normal, undamaged structure
- **Non-self patterns** = Damaged structure

🏗️ 2. What is Structural Damage Classification?

This involves:

- Monitoring structural data (like vibrations, strain, displacement)
- Detecting **anomalies** or **deviations** from healthy behavior
- Classifying the type or severity of damage

Common input features may come from **sensor data** such as:

- Natural frequency changes
- Mode shape deviations
- Acoustic emissions
- Strain readings

⚙️ 3. Steps to Apply Artificial Immune Pattern Recognition for Structure Damage Classification

---

Step 1: Data Collection

- Collect sensor data from the structure under different conditions: normal and various damaged states.

- Use datasets of vibration modes, frequencies, strain measurements, etc.

Step 2: Encoding (Antibody and Antigen Representation)

- Each data pattern is represented as a **vector**.
- **Self-patterns** (healthy structure) → used to create **memory cells or detectors**.
- **Antigens** → new incoming patterns to be classified as healthy/damaged.

Step 3: Detector Generation (Negative Selection Algorithm)

- Generate a population of detectors that do **not match self** data (i.e., detectors are trained not to recognize healthy data).
- These detectors act like "immune cells" that only respond to foreign (damaged) inputs.

Step 4: Clonal Selection and Mutation (Learning)

- When a detector successfully identifies a damaged pattern, it is cloned and mutated to improve detection.
- The best detectors are retained and refined over time.

Step 5: Classification

- When new structural data is input, the trained immune system:
    - Compares it against the memory set (healthy patterns)
    - Uses affinity measures (e.g., Euclidean distance) to determine if it's similar to "self" or "non-self"
- Based on this, the pattern is **classified as damaged or not** (binary classification) or **type of damage** (multi-class classification).

Step 6: Update Memory Set

- Periodically, the memory set is updated with new data to adapt to evolving damage patterns or structural aging.

✅ **Advantages of AIPR in Damage Classification**

- Works well with **incomplete or noisy data**
- Adaptable to **changing environments**
- Effective for **anomaly detection** in **nonlinear systems**
- Doesn't require huge training data compared to deep learning

📊 Real-World Example (Simplified)

Let's say you have a bridge equipped with vibration sensors. Over time, the bridge might suffer:

- **No damage** → self pattern
- **Minor crack** → class 1
- **Major fatigue** → class 2

You train your immune-inspired system with:

- Normal readings → to create memory cells
- Then use detectors to classify incoming sensor data based on how similar/dissimilar it is to the memory set.

🧬 Related AIS Algorithms Used

AIS Technique	Usage in Damage Classification
Negative Selection	For anomaly detection (detect "non-self" patterns)
Clonal Selection	For learning and improving damage classification
Immune Network Models	For memory and recognition of complex damage patterns

CODE EXPLAINATION

📦 Import Libraries

```
import numpy as np
from sklearn.preprocessing import StandardScaler
from scipy.stats import mode
from sklearn.datasets import make_classification
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

- numpy: For numerical operations and arrays.
- StandardScaler: Standardizes the dataset (mean=0, variance=1).
- mode: Finds the most frequent label in a set (used for majority voting).

- make_classification: Generates synthetic classification datasets.
- matplotlib.pyplot: For plotting.
- PCA: Reduces dimensionality for visualization (from 10D to 2D).

🧪 Generate Synthetic Dataset

```
data, labels = make_classification(n_samples=100, n_features=10,
                                   n_informative=5, n_redundant=0,
                                   n_clusters_per_class=1, random_state=42)
```

- Generates 100 samples, each with 10 features.
- 5 features are informative for classification.
- 2 classes (labels are 0 or 1).

✂️ Split Dataset

```
split = int(0.8 * len(data))
X_train, X_test = data[:split], data[split:]
y_train, y_test = labels[:split], labels[split:]
```

- Splits 80% of the data for training (X_train, y_train) and 20% for testing.

🔄 Normalize Features

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

- Scales data to have mean 0 and variance 1.
- Important for distance-based models like nearest neighbor.

📌 Select Detectors

```
idx = np.random.choice(len(X_train), 30, replace=False)
detectors = X_train[idx]
```

detector_labels = y_train[idx]

- Randomly selects 30 samples from training data to act as **detectors**.
- These simulate **antibodies** detecting “foreign patterns.”

🧠 Predict Using 3-Nearest Detectors

```
predictions = []
for x in X_test:
    dists = np.linalg.norm(detectors - x, axis=1)
    k_idx = np.argsort(dists)[:3]
    k_labels = detector_labels[k_idx]
    predictions.append(mode(k_labels, keepdims=True)[0][0])
predictions = np.array(predictions)
```

- For each test sample:
    1. Computes Euclidean distance to all detectors.
    2. Selects the 3 closest detectors (k=3).
    3. Uses **majority voting** (mode) from the labels of those 3 detectors.
- Appends the predicted label to predictions.

📊 Evaluate Accuracy

```
acc = np.mean(predictions == y_test)
print(f"Accuracy: {acc:.2f}")
```

- Compares predicted labels to actual labels.
- Prints classification accuracy.

📈 Visualization Section

📜 Dimensionality Reduction with PCA

```
pca = PCA(n_components=2)
X_train_2D = pca.fit_transform(X_train)
```

X_test_2D = pca.transform(X_test)

detectors_2D = pca.transform(detectors)

- Reduces 10D data to 2D using **Principal Component Analysis** (PCA).
- Allows us to plot data in 2D space.

🖼️ Plot the Data

plt.figure(figsize=(10, 6))

- Sets the size of the plot.

🎨 Plot Training Data

plt.scatter(X_train_2D[:, 0], X_train_2D[:, 1], c=y_train, cmap='gray', label='Training Data', alpha=0.5)

- Plots all training points in **gray**.
- alpha=0.5 makes them semi-transparent.

❌ Plot Detectors

plt.scatter(detectors_2D[:, 0], detectors_2D[:, 1], c=detector_labels, cmap='cool', edgecolor='black', label='Detectors', s=100, marker='X')

- Plots the 30 detectors using **X markers** with a distinct color map (cool).
- Labels correspond to class (0 or 1).

🧪 Plot Test Predictions

plt.scatter(X_test_2D[:, 0], X_test_2D[:, 1], c=predictions, cmap='spring', label='Test Predictions', marker='o', s=60)

- Plots test data using predicted labels.
- Uses a different color map (spring) for visibility.

📝 Finalize the Plot

```
plt.title('Immune-Inspired Classifier Visualization (PCA Projection)')
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

- Adds title, axis labels, legend, and shows the graph.

📌 Summary of the Visualization

- **Gray dots**: Training data (real samples).
- **X markers**: Detectors chosen randomly from training data.
- **Colored circles**: Test points labeled by the immune-inspired classifier.
- PCA is used to reduce dimensionality so you can see all points in 2D.

GRAPH

This graph is a **2D visualization** of your **immune-inspired classifier** using **PCA (Principal Component Analysis)** to project high-dimensional data into 2D space. Let's break it down visually:

📈 Graph Title

"Immune-Inspired Classifier Visualization (PCA Projection)"

- Indicates that PCA is used to reduce your original 10-dimensional feature space to 2D for visualization.

🔍 Axes

- **X-axis: PCA 1**
- **Y-axis: PCA 2**

These are the first two **principal components**, which capture the most variance (or spread) in your data.

🟠 Data Types Shown

The plot displays three key components:

1. Training Data (Gray Circles)

- Represent the original 80 training samples.
- Their colors are not showing class labels explicitly (as grayscale is used), but they form the **overall structure of the dataset**.

2. Detectors (X Markers)

- Colored using the **'cool' colormap** (often magenta and cyan shades).
- These are **30 randomly selected points** from the training set.
- They act like **"immune cells"** detecting incoming unknown inputs.
- Their class labels determine how they classify test points.

3. Test Predictions (Yellow and Pink Circles)

- Represent test samples classified by your model.
- Their colors (from the **'spring' colormap**, usually yellow and pink) reflect the **predicted classes**.
- These are NOT colored based on the true label — just the prediction.

🔍 How to Interpret

- Look at **clusters**:
    - If yellow and pink predictions are grouped closely with detectors of the same color → good classification.
    - If they are spread randomly → poor classification.
- **Overlay and proximity**:
    - Detectors close to test points influence the prediction due to nearest-neighbor logic.
    - If a test point lies in the region dominated by pink detectors, it's more likely classified as pink (and vice versa).

✅ **Usefulness of This Graph**

- Helps **visually understand** how detectors are spread.
- Shows **how well the classifier generalizes** test data.
- Allows you to **inspect overlaps or ambiguities** where predictions may be uncertain.

8. Implement DEAP (Distributed Evolutionary Algorithms) using Python.

DEAP (Distributed Evolutionary Algorithms in Python) is a powerful evolutionary algorithm framework in Python that supports the development of evolutionary algorithms (EAs), such as Genetic Algorithms (GAs), Genetic Programming (GP), and other optimization techniques.

It is built on the concept of evolutionary algorithms, which simulate the process of natural evolution to solve optimization problems. DEAP provides the tools to build and run evolutionary algorithms in a parallel, distributed, and efficient manner, making it a flexible framework for both beginners and advanced users.

Here's a comprehensive explanation of DEAP:

1. Introduction to Evolutionary Algorithms

Evolutionary algorithms are inspired by the process of natural selection and evolution. They are heuristic search algorithms used for optimization problems, where the goal is to find the best solution from a set of possible solutions. In evolutionary algorithms, candidate solutions are evolved through generations using operators like selection, crossover, mutation, and reproduction. Some common types of evolutionary algorithms are:

- **Genetic Algorithms (GA):** These algorithms are based on the principles of natural genetics, using selection, crossover, and mutation to evolve solutions.
- **Genetic Programming (GP):** A form of evolutionary computation where computer programs (usually trees) evolve to perform a specific task.
- **Differential Evolution (DE):** A population-based algorithm for real-valued optimization problems.
- **Evolution Strategies (ES):** Used for optimization problems that focus on continuous search spaces.

2. DEAP Framework Overview

DEAP is a versatile Python library that simplifies the development and experimentation with evolutionary algorithms. It provides a flexible and highly extensible system for optimization and machine learning.

Some key features of DEAP include:

- **Simple API:** DEAP has a very user-friendly API that allows you to define evolutionary algorithms in a simple and straightforward way.
- **Modular Design:** It is built to be modular, allowing users to select and customize components, such as selection methods, crossover, mutation, and fitness evaluation functions.
- **Parallel and Distributed Computing Support:** DEAP supports parallelism and distributed computing, which speeds up the evolutionary algorithm process, especially when dealing with large datasets or complex tasks.
- **Extensive Documentation:** DEAP provides detailed documentation and tutorials to help users get started quickly.

3. DEAP Components

DEAP consists of several main components that are combined to form an evolutionary algorithm. These components include:

a) Individual:

An individual is a candidate solution to the optimization problem. In DEAP, an individual can be represented by a list, a string, or even a more complex data structure like a tree or graph (for genetic programming).

b) Population:

A population consists of multiple individuals. The evolutionary algorithm evolves this population over several generations to improve the solutions.

c) Fitness:

The fitness function evaluates the quality of the individuals in the population. It is problem-specific, and the goal is to maximize or minimize the fitness value (depending on the problem being solved).

d) Selection:

Selection is the process by which individuals are chosen for reproduction. It is typically based on the fitness of the individuals, with more fit individuals having a higher chance of being selected.

e) Crossover (Recombination):

Crossover is the process where two parent individuals combine their genetic material to create offspring. This operation simulates the genetic recombination that happens during sexual reproduction in nature.

f) Mutation:

Mutation introduces random changes to an individual's genetic material. It is used to maintain genetic diversity in the population and explore new regions of the search space.

g) Replacement:

Replacement is the process of selecting which individuals are kept in the next generation. Some algorithms replace the worst individuals, while others use elitism (keeping the best individuals).

6. Example Use Cases of DEAP

- **Optimization Problems:** DEAP can be used for various optimization problems, including combinatorial optimization, continuous optimization, multi-objective optimization, and more.
- **Machine Learning:** DEAP can be used to optimize machine learning models, such as tuning hyperparameters or evolving neural networks.
- **Game Strategies:** DEAP can be used to evolve strategies for games and simulations.
- **Artificial Life and Genetic Programming:** DEAP is useful in artificial life simulations and evolving programs for specific tasks.

7. Conclusion

DEAP provides a comprehensive framework for working with evolutionary algorithms. With its flexible design and extensive set of tools, DEAP enables easy implementation of complex algorithms for various types of optimization problems. By leveraging its support for parallelism, DEAP can be an efficient tool for solving large-scale problems in an evolutionary way.

This Python code implements a **Genetic Algorithm (GA)** using the **DEAP (Distributed Evolutionary Algorithms in Python)** library to solve a simple problem:

Maximize the number of 1s in a binary list of length 10.

Let's go step by step to understand the entire code.

🧠 **Objective**

We want to **evolve a population of binary individuals** (lists of 0s and 1s), such that the final solution contains **as many 1s as possible** — that is, a list like [1,1,1,1,1,1,1,1,1,1].

🧩 **Step-by-Step Breakdown**

📌 Step 1: Define Fitness & Individual Types

```
creator.create("FitnessMax", base.Fitness, weights=(1.0,))
creator.create("Individual", list, fitness=creator.FitnessMax)
```

- FitnessMax: A DEAP fitness class that indicates **we want to maximize** the fitness (instead of minimizing).
- Individual: An **individual solution**, which is just a Python list that carries a fitness attribute.

🧰 Step 2: Define Toolbox – How to Create Individuals and Populations

```
toolbox.register("attr_bool", random.randint, 0, 1)
toolbox.register("individual", tools.initRepeat, creator.Individual, toolbox.attr_bool, 10)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
```

- attr_bool: A function that generates either 0 or 1 (random gene).
- individual: Creates an individual of **10 binary genes**.
- population: A list of individuals. This will be the population.

⚙️ Step 3: Define Genetic Operators and Fitness Function

```
def eval_fitness(individual):
    return sum(individual),  # Comma makes it a tuple

toolbox.register("evaluate", eval_fitness)
toolbox.register("mate", tools.cxTwoPoint)  # Crossover: swap sections of two individuals
toolbox.register("mutate", tools.mutFlipBit, indpb=0.1)  # Flip bit with 10% chance
toolbox.register("select", tools.selTournament, tournsize=3)  # Select best of 3 randomly
```

- eval_fitness: The more 1s in the individual, the higher the fitness.
- mate: Two-point crossover, a typical genetic operation.
- mutate: Flips each gene with a probability of 10%.

- select: Tournament selection chooses the best individual among 3 randomly picked ones.

🚀 Step 4: Run the Evolutionary Algorithm

```
def run_ea():
    pop = toolbox.population(n=30)  # Initial population of 30 individuals
    hof = tools.HallOfFame(1)  # Stores the best individual
    stats = tools.Statistics(lambda ind: ind.fitness.values)
    stats.register("max", max)  # Track maximum fitness in each generation

    pop, log = algorithms.eaSimple(pop, toolbox, cxpb=0.5, mutpb=0.2, ngen=10,
                                   stats=stats, halloffame=hof, verbose=True)

    print("\nBest individual:", hof[0])
    print("Fitness:", hof[0].fitness.values[0])
```

- eaSimple(): Runs the evolutionary algorithm.
    - cxpb=0.5: 50% chance to crossover.
    - mutpb=0.2: 20% chance for mutation.
    - ngen=10: Run for 10 generations.
- stats: Logs the **maximum fitness** in each generation.
- halloffame: Stores the best solution found throughout all generations.

📌 Final Output

After running the algorithm:

```
run_ea()
```

You'll see output like:

```
gen     nevals  max
0   30      8
```

1 19 9

...

9 15 10

Best individual: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

Fitness: 10

This shows the **evolution of solutions** and ends with the **best individual** that has maximum 1s.

✅ **Summary**

- The code uses **DEAP** to evolve solutions that maximize the number of 1s in a binary list.
- It demonstrates basic GA concepts: **selection**, **crossover**, **mutation**, and **fitness-based survival**.
- Very useful as a **template** for more complex optimization problems!

9. 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.

Ant Colony Optimization (ACO) is a **bio-inspired optimization algorithm** that mimics the **foraging behavior of real ants** to solve complex problems, especially **combinatorial optimization problems** like the **Traveling Salesman Problem (TSP)**, **scheduling**, **routing**, and more.

🐜 **What is the Idea Behind ACO?**

In nature, ants find the **shortest path** between their colony and a food source by:

1. **Wandering randomly** at first.
2. **Depositing pheromones** on the paths they travel.
3. **Following paths with stronger pheromone trails**, which usually indicate better (shorter) routes.

4. **Reinforcing** good paths as more ants follow them.

Over time, **the shortest path has the most pheromone**, and ants are more likely to choose it.

💡 How Does ACO Work in Computing?

ACO uses **artificial ants** to simulate this behavior in a computer algorithm:

1. **Problem is modeled as a graph** (nodes = states/cities, edges = transitions/paths).
2. **Ants** are placed on the graph and build **solutions step-by-step**.
3. Each decision is **probabilistic**, based on:
    - **Pheromone levels** (learned experience)
    - **Visibility** (e.g., closeness, cost, or time)
4. After all ants complete their solution:
    - **Pheromones are updated**:
        - **Evaporation** reduces old pheromones.
        - **Reinforcement** adds pheromones to good solutions.
5. **Repeat for many generations** until the best solution is found.

✨ Features of ACO

Feature	Description
Stochastic	Uses probability, not deterministic rules
Distributed	Multiple agents (ants) work in parallel
Positive feedback	Good paths get more pheromones
Heuristic	Uses problem-specific information (like distances)

🧠 ACO is Good For:

- **Traveling Salesman Problem (TSP)**
- **Vehicle Routing**

- **Job Scheduling**
- **Network Routing**
- **Knapsack problem**
- **Maze solving**, etc.

✅ Summary

Ant Colony Optimization (ACO) is a powerful, nature-inspired algorithm that:

- **Solves optimization problems** by simulating how ants find the best path.
- **Uses pheromone trails** and **probabilistic decisions** to explore and exploit possible solutions.
- **Improves over time** by reinforcing better solutions and forgetting poor ones.

CODE EXPLAINATION

This code implements **Ant Colony Optimization (ACO)** to solve the **Traveling Salesman Problem (TSP)** and includes **visualization** of the best tour found. Let's break it down **step by step**:

🧠 Goal

Find the shortest route that visits each city **once** and returns to the starting city.

🔧 Step-by-Step Code Explanation

✅ 1. City Coordinates

cities = {0: (0, 0), 1: (1, 5), 2: (5, 2), 3: (6, 6), 4: (8, 3)}

You define 5 cities with their (x, y) coordinates.

✅ 2. Distance and Pheromone Initialization

distance = {(i, j): math.dist(cities[i], cities[j]) for i in cities for j in cities if i != j}

pheromone = {edge: 1.0 for edge in distance}

- distance: Stores the Euclidean distance between every pair of cities.
- pheromone: Each edge (city pair) starts with an initial pheromone level of 1.0.

✅ 3. ACO Parameters

num_ants = 10

num_iters = 100

alpha = 1.0 # influence of pheromone

beta = 5.0   # influence of distance (visibility)

evaporation = 0.5

Q = 100   # pheromone deposit constant

These parameters control:

- How **ants make decisions**
- How **pheromones are updated**
- How **much exploration vs. exploitation** happens

✅ 4. Choosing the Next City

```
def choose_next(curr, visited):
    probs = []
    for c in cities:
        if c not in visited:
            pher = pheromone[(curr, c)] ** alpha
            visibility = (1 / distance[(curr, c)]) ** beta
            probs.append((c, pher * visibility))
```

This function:

- Calculates **probabilities** of going to each **unvisited** city
- Considers both:
    - **Pheromone trail** (how attractive an edge is)

- **Visibility** (closer cities are more attractive)

✅ 5. Building a Tour

```
def build_tour():
    path = [random.choice(list(cities))]
    visited = set(path)
    while len(path) < len(cities):
        next_city = choose_next(path[-1], visited)
        path.append(next_city)
        visited.add(next_city)
    return path + [path[0]]  # return to start
```

Each **ant builds a complete path** by:

- Starting at a **random city**
- Visiting **all other cities** based on probability
- Returning to the **start**

✅ 6. Main ACO Loop

```
for _ in range(num_iters):
    all_tours = []
    for _ in range(num_ants):
        tour = build_tour()
        length = sum(distance[(tour[i], tour[i + 1])] for i in range(len(tour) - 1))
```

- num_iters: The algorithm runs multiple **generations**
- In each iteration:
    - Each **ant builds a tour**
    - Distance of each tour is calculated
    - The **shortest path so far** is updated

✅ 7. Pheromone Update

```
for edge in pheromone:
    pheromone[edge] *= (1 - evaporation)


for path, length in all_tours:
    for i in range(len(path) - 1):
        a, b = path[i], path[i + 1]
        pheromone[(a, b)] += Q / length
        pheromone[(b, a)] += Q / length  # undirected
```

- **Evaporation**: Reduces all pheromone values to avoid early convergence
- **Deposit**: Each ant deposits pheromone on its path—**shorter tours deposit more**

✅ 8. Output Best Tour

```
print("\nBest Tour Found:")
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_length:.2f}")
```

This prints the **shortest tour found** and its total distance.

✅ 9. Visualization with matplotlib

```
def plot_tour(path):
    x = [cities[i][0] for i in path]
    y = [cities[i][1] for i in path]

    plt.figure(figsize=(8, 6))
    plt.plot(x, y, 'bo-', label="Best Tour Path", linewidth=2, markersize=8)
```

- Plots all city coordinates
- Connects them based on the **best path**
- Adds labels for clarity

🎯 **Summary**

Component	Purpose
cities	Input points for the TSP
pheromone	Memory system of ants
choose_next()	Probabilistic path decision
build_tour()	Constructs full paths using ants
ACO Loop	Repeated search and pheromone updates
plot_tour()	Visual output of final optimized solution

Would you like a version that **animates the search process** over time as ants explore?

10. Create and Art with Neural style transfer on given image using deep learning.

Sure! Let's dive into the **theory behind Neural Style Transfer (NST)** — a fascinating deep learning technique that creates **art** by combining the *content* of one image with the *style* of another.

🎨 **What Is Neural Style Transfer?**

Neural Style Transfer (NST) is a technique that **generates a new image** by blending:

- The **content** of a base image (e.g., a photo of a person or place), and
- The **style** of another image (e.g., a Van Gogh painting).

It was introduced by **Gatys et al. in 2015**, and it uses **convolutional neural networks (CNNs)** to separate and recombine content and style from images.

🔍 **How It Works: Step-by-Step**

1. Use a Pretrained CNN

NST uses a pretrained model like **VGG19** or **MobileNetV2**, which is trained on ImageNet, to extract features from images. CNNs naturally break images down into layers of:

- **Low-level features** (edges, colors)
- **High-level features** (object shapes and layout)

---

2. Define the Inputs

- **Content image**: The image whose **structure** and layout you want to keep.
- **Style image**: The image whose **color**, **brushstrokes**, and **textures** you want to apply.
- **Target image**: A copy of the content image that will be optimized to blend both content and style.

---

3. Extract Features

Using CNN layers:

- **Content features** are extracted from higher layers (they capture layout/structure).
- **Style features** are extracted from lower and mid-level layers using **Gram matrices**, which capture texture and style patterns.

---

4. Calculate Losses

There are **three key losses** used:

a. Content Loss

Ensures that the **target image has the same structure** as the content image.

$$\text{Content Loss} = \|F_{\text{target}} - F_{\text{content}}\|^2$$

b. Style Loss

Ensures that the **target image mimics the textures** of the style image using **Gram matrices**:

$$\text{Style Loss} = \sum_l \|G_l^{\text{target}} - G_l^{\text{style}}\|^2$$

where G is the Gram matrix of features at layer l.

c. Total Loss

$$\text{Total Loss} = \alpha \cdot \text{Content Loss} + \beta \cdot \text{Style Loss}$$

- α: weight for content
- β: weight for style (usually much higher than α\alphaα)

5. Optimization

The target image is **updated pixel-by-pixel** using gradient descent (e.g., with Adam optimizer) to **minimize the total loss**. The neural network itself is **not trained** — the **image is optimized**, not the model.

🖼️ **Final Result**

You get an image that:

- Has the **shapes and structure** of the content image,
- And the **colors, textures, and strokes** of the style image.

✅ **Summary Table**

Term	Description
Content Image	Source structure/image (e.g., landscape photo)
Style Image	Artistic reference (e.g., Picasso painting)
Gram Matrix	Captures texture and style via feature correlations
Content Loss	Keeps layout similar to the content image
Style Loss	Mimics artistic style in the result
Optimization	Gradually improves the target image via backprop

CODE EXPLANATION

This code performs **Neural Style Transfer (NST)** using **PyTorch** and **MobileNetV2**, where it blends the **content** of one image with the **style** of another to generate artistic output. Let's walk through the code step by step.

🧠 **High-Level Goal**

Create an image that:

- **Looks like** the content image (base.jpg)
- **Feels like** the style image (style.jpg)

🔧 **Step-by-Step Code Breakdown**

🔹 **1. Import Libraries**

```
import torch
import torch.nn.functional as F
from torchvision import models, transforms
```

```
from PIL import Image
import matplotlib.pyplot as plt
```

- torch: For tensor and model operations
- torchvision.models: Pre-trained CNN (MobileNetV2)
- transforms: Preprocess input images
- PIL: Image loading
- matplotlib: Display images

🔹 2. Load and Preprocess Image

```
def load_image(img_path, size=256):
    image = Image.open(img_path).convert('RGB')
    transform = transforms.Compose([
        transforms.Resize((size, size)),
        transforms.ToTensor()
    ])
    return transform(image).unsqueeze(0)
```

- Loads image from disk and converts it into a **tensor** suitable for deep learning.
- unsqueeze(0) adds a batch dimension → shape becomes [1, 3, 256, 256].

🔹 3. Gram Matrix Function

```
def gram_matrix(tensor):
    b, c, h, w = tensor.size()
    features = tensor.view(c, h * w)
    G = torch.mm(features, features.t())
    return G.div(c * h * w)
```

- Computes the **Gram matrix**, which captures the **style** (texture patterns) of the image.
- Used to compare style between target and style images.

🔹 4. Feature Extraction Model

```
class FeatureExtractor(torch.nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.model = models.mobilenet_v2(weights=models.MobileNet_V2_Weights.IMAGENET1K_V1).features
        self.layers = layers

    def forward(self, x):
        features = []
        for i, layer in enumerate(self.model):
            x = layer(x)
            if str(i) in self.layers:
                features.append(x)
        return features
```

- Uses **MobileNetV2** to extract features.
- Only features from specific layers ('4', '7') are returned for style and content representation.

🔹 5. Load and Display Images

```
content = load_image('base.jpg').to(device)
style = load_image('style.jpg').to(device)
target = content.clone().requires_grad_(True)
```

- content: Image whose structure should be preserved.
- style: Image whose texture (style) should be applied.
- target: The image we will optimize to combine content and style.

```
imshow(content, "Content Image")
imshow(style, "Style Image")
```

Displays both input images.

🔹 6. Feature Extraction

```
extractor = FeatureExtractor(layers=['4', '7']).to(device).eval()


with torch.no_grad():
    style_features = extractor(style)
    content_features = extractor(content)
    style_grams = [gram_matrix(f) for f in style_features]
```

- Extracts feature maps from both images.
- style_grams: Used to compare texture/style later.

🔹 7. Optimizer Setup

```
optimizer = torch.optim.Adam([target], lr=0.01)
```

- Adam optimizer updates the target image directly.

🔹 8. Neural Style Transfer Loop

```
for step in range(200):
    target_features = extractor(target)
    content_loss = F.mse_loss(target_features[-1], content_features[-1])
    style_loss = sum(F.mse_loss(gram_matrix(f), g) for f, g in zip(target_features, style_grams))
    loss = content_loss + 1e5 * style_loss
```

- **Content Loss**: MSE between content features and target features.
- **Style Loss**: MSE between Gram matrices of style and target.
- 1e5: Large weight to emphasize style in final result.

```
    optimizer.zero_grad()
    loss.backward()
```

optimizer.step()

- Backpropagation: Gradients are computed with respect to the **target image**
- Optimizer updates pixel values in target to reduce both losses

---

🔹 9. Display Final Image

```
output = target.squeeze().detach().cpu().clamp(0, 1).permute(1, 2, 0)
plt.imshow(output)
plt.axis('off')
plt.title("Stylized Output")
plt.show()
```

- Removes batch dimension
- Clamps values between 0 and 1 (valid pixel range)
- Converts tensor to image format and displays the result

✅ Summary

Component	Purpose
load_image	Load and resize image
gram_matrix	Capture style (texture)
FeatureExtractor	Extract relevant CNN layers
optimizer	Update the image directly using backpropagation
content_loss	Preserve structure
style_loss	Apply artistic patterns
imshow()	Show results visually