**Name: -** Shivam Borse

**Roll No.: -** BEAD20119

**Subject: -** CL III (Computational Intelligence)

**Assignment No. 04**

**Problem Statement:** 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

**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)

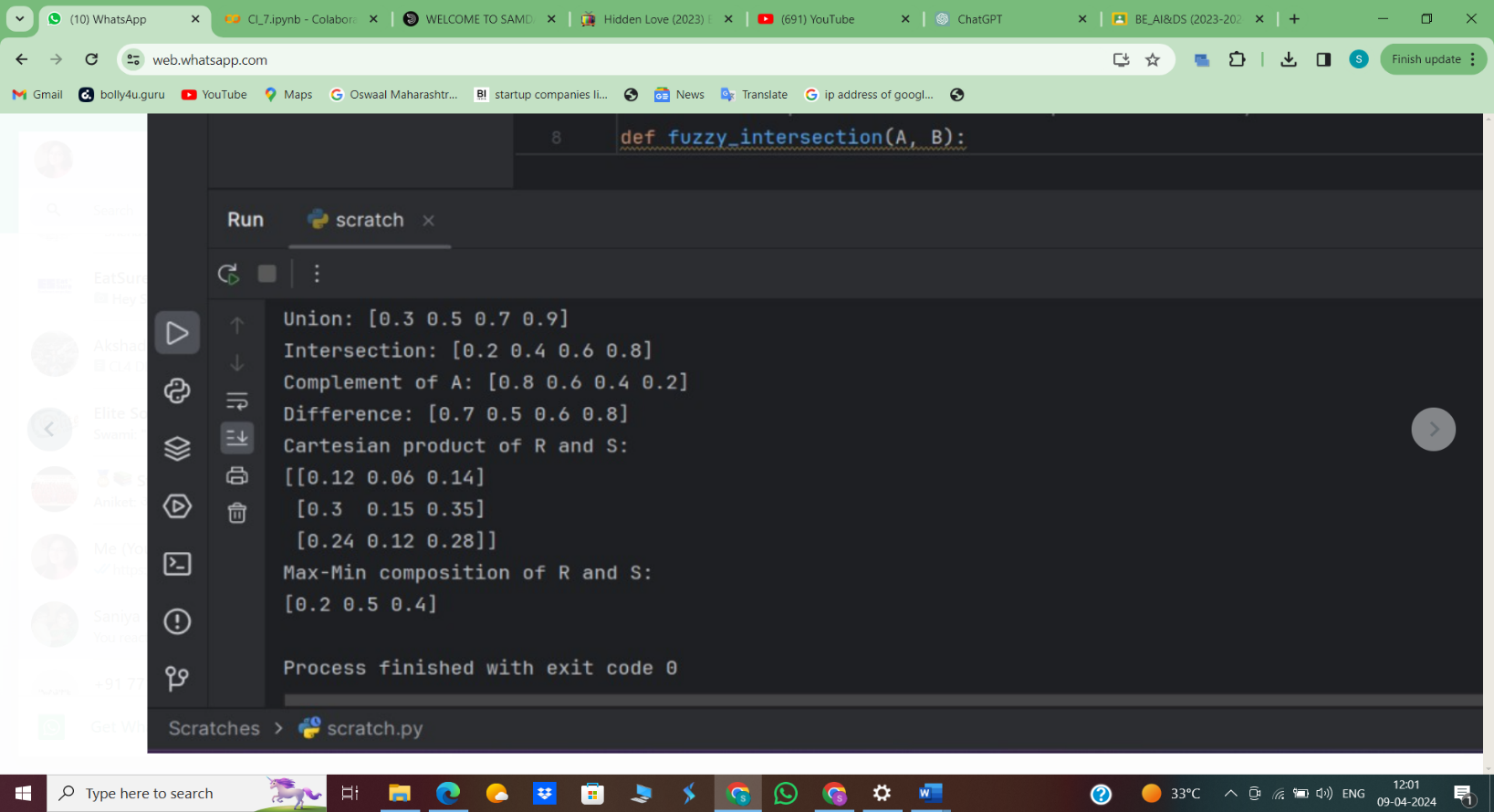
print("Cartesian product of R and S:")

print(cartesian\_result)

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

print(composition\_result)

**Output:-**

****

**Name: -** Shivam Borse

**Roll No.: -** BEAD20119

**Subject: -** CL III (Computational Intelligence)

**Assignment No. 05**

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

**Code-**pip install deap

import random

from deap import base, creator, tools, algorithms

# Define evaluation function (this is a mock function, replace this with your actual evaluation

# function)

def evaluate(individual):

    # Here 'individual' represents the parameters for the neural network

    # You'll need to replace this with your actual evaluation function that trains the neural network

    # and evaluates its performance

    # Return a fitness value (here, a random number is used as an example)

    return random.random(),

# Define genetic algorithm parameters

POPULATION\_SIZE = 10

GENERATIONS = 5

# Create types for fitness and individuals in the genetic algorithm

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

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

# Initialize toolbox

toolbox = base.Toolbox()

# Define attributes and individuals

toolbox.register("attr\_neurons", random.randint, 1, 100)  # Example: number of neurons

toolbox.register("attr\_layers", random.randint, 1, 5)  # Example: number of layers

toolbox.register("individual", tools.initCycle, creator.Individual, (toolbox.attr\_neurons,

                                                                      toolbox.attr\_layers), n=1)

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

# Genetic operators

toolbox.register("evaluate", evaluate)

toolbox.register("mate", tools.cxTwoPoint)

toolbox.register("mutate", tools.mutUniformInt, low=1, up=100, indpb=0.2)

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

# Create initial population

population = toolbox.population(n=POPULATION\_SIZE)

# Run the genetic algorithm

for gen in range(GENERATIONS):

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

    fitnesses = toolbox.map(toolbox.evaluate, offspring)

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

        ind.fitness.values = fit

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

# Get the best individual from the final population

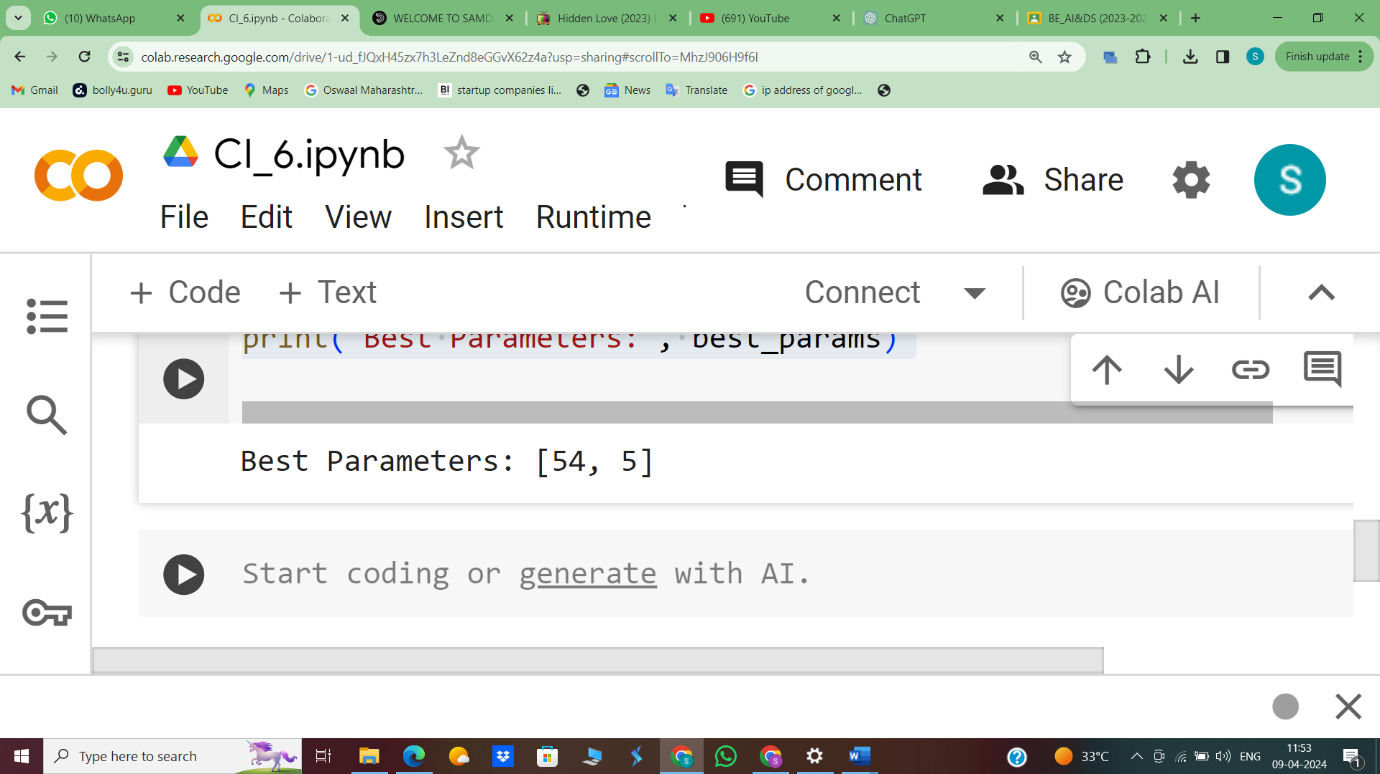
best\_individual = tools.selBest(population, k=1)[0]

best\_params = best\_individual

# Print the best parameters found

print("Best Parameters:", best\_params)

**Output:-**



**Name: -** Shivam Borse

**Roll No.: -** BEAD20119

**Subject: -** CL III (Computational Intelligence)

**Assignment No. 06**

**Problem Statement:** Implementation of Clonal selection algorithm using Python.

**Code-**

import random

import numpy as np

# Define the objective function

def objective\_function(x):

# Example: Sphere function

return sum([i\*\*2 for i in x])

# Initialize population

def initialize\_population(pop\_size, dimensions, lower\_bound, upper\_bound):

population = []

for \_ in range(pop\_size):

individual = np.random.uniform(lower\_bound, upper\_bound, dimensions)

population.append(individual)

return population

# Clone an individual

def clone\_individual(individual, clone\_factor):

clones = []

for \_ in range(clone\_factor):

clone = np.array(individual)

for i in range(len(clone)):

mutation\_rate = random.uniform(0, 1)

if mutation\_rate < 0.5:

clone[i] += random.uniform(-1, 1)

clones.append(clone)

return clones

# Select the best individuals

def select\_best(population, num\_selected, num\_clones, clone\_factor):

population.sort(key=lambda x: objective\_function(x))

selected = []

for i in range(min(len(population), num\_selected)):

clones = clone\_individual(population[i], clone\_factor)

selected.extend(clones)

return selected[:num\_clones]

# Main function for clonal selection algorithm

def clonal\_selection\_algorithm(pop\_size, dimensions, lower\_bound, upper\_bound,

num\_generations, num\_selected, num\_clones, clone\_factor):

population = initialize\_population(pop\_size, dimensions, lower\_bound, upper\_bound)

for \_ in range(num\_generations):

selected = select\_best(population, num\_selected, num\_clones, clone\_factor)

population = selected

best\_solution = min(population, key=lambda x: objective\_function(x))

return best\_solution, objective\_function(best\_solution)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

pop\_size = 100

dimensions = 2

lower\_bound = -5.0

upper\_bound = 5.0

num\_generations = 100

num\_selected = 10

num\_clones = 5

clone\_factor = 3

best\_solution, best\_fitness = clonal\_selection\_algorithm(pop\_size,dimensions, lower\_bound,upper\_bound, num\_generations, num\_selected, num\_clones, clone\_factor)

print("Best solution:", best\_solution)

print("Best fitness:", best\_fitness)

**Output:-**

