SCT PRACTICAL

PRAC 1 : Design a simple linear neural network model

CODE:

x = float(input("Enter value of x:"))

w = float(input("Enter value of weight w:"))

b = float(input("Enter value of bias b:"))

net = int(w \* x + b)

if net < 0:

    out = 0

elif 0 <= net <= 1:

    out = net

else:

    out = 1

print("net =", net)

print("output =", out)

PRAC 1B: Calculate the output of neural net using both binary and bipolar sigmoidal function

COD3:

import math

bias = float(input("Enter the value of bias: "))

n = int(input("Enter the number of input neurons: "))

w = [ ]

x = [ ]

for i in range(0,n):

a = float(input("Enter the input: "))

x.append(a)

b = float(input("Enter the weight: "))

w.append(b)

print("The given weights are: ")

print(w)

print("The given input are: " )

print(x)

y = bias

for i in range(0,n):

y = y + (w[i]\*x[i])

print("The calculated net input y : ")

print(y)

# Applying Binary Sigmoidal function on the net input i.e y

binary = 1/(1+ (math.exp(-y)))

print("The output after applying binary sigmoidal function activation ")

print (round(binary, 3))

# Applying Bipolar Sigmoidal function on the net input i.e y

bipolar = -1+(2/(1+ (math.exp(-y))))

print("The output after applying bipolar sigmoidal function activation ")

print(round(bipolar, 3))

PRACTICAL 2A: Generate AND/NOT function using McCulloch-Pitts neural net.

CODE:

#Pract 2A

num\_ip = int(input("Enter the number of inputs : "))

theta = 1

x1 =[]

x2 =[]

for i in range(0,num\_ip):

a = int(input("Enter the input x1: "))

x1.append(a)

b = int(input("Enter the input x2: "))

x2.append(b)

print("x1 = ", x1)

print("x2 = ", x2)

print("Value of theta is 1.")

print("Case 1: For calculating the net input the weight is considered as w1 = w2 = 1")

w1 = w2 = 1

case\_y1 =[]

case\_yin1 = []

print("x1 w1 x2 w2 case\_y1 case\_yin1")

for i in range(0,num\_ip):

case\_y1.append(x1[i]\*w1 + x2[i]\*w2)

if (case\_y1[i] >= theta):

case\_yin1.append( 1 )

else:

case\_yin1.append( 0 )

print(x1[i]," ", w1," ",x2[i]," ",w2," ", case\_y1[i]," ",case\_yin1[i])

print("From the calculated net inputs, it is not possible to fire the neuron form input (1, 0) only."

"\n Hence, these weights are not suitable. ")

print("Case 2: For calculating the net input the weight is considered as w1 = 1, w2 = -1")

w1 = 1

w2 = -1

case\_y2 =[]

case\_yin2 = []

print("x1 w1 x2 w2 case\_y2 case\_yin2")

for i in range(0,num\_ip):

case\_y2.append(x1[i]\*w1 + x2[i]\*w2)

if (case\_y2[i] >= theta):

case\_yin2.append( 1 )

else:

case\_yin2.append( 0 )

print(x1[i]," ", w1," ",x2[i]," ",w2," ", case\_y2[i]," ",case\_yin2[i])

print("From the calculated net inputs, it is possible to fire the neuron form input (1, 0) only."

"\n Hence, these weights are suitable. ")

PRAC 2B:

CODE: Generate XOR function using McCulloch-Pitts neural net.

#Pract 2B

import numpy as np

print('Ente1r weights')

w11=int(input('Weight w11='))

w12=int(input('weight w12='))

w21=int(input('Weight w21='))

w22=int(input('weight w22='))

v1=int(input('weight v1='))

v2=int(input('weight v2='))

print('Enter Threshold Value')

theta=int(input('theta='))

x1=np.array([0, 0, 1, 1])

x2=np.array([0, 1, 0, 1])

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

con=1

y1=np.zeros((4,))

y2=np.zeros((4,))

y=np.zeros((4,))

while con==1:

zin1=np.zeros((4,))

zin2=np.zeros((4,))

zin1=x1\*w11+x2\*w21

zin2=x1\*w21+x2\*w22

print("z1",zin1)

print("z2",zin2)

for i in range(0,4):

if zin1[i]>=theta:

y1[i]=1

else:

y1[i]=0

if zin2[i]>=theta:

y2[i]=1

else:

y2[i]=0

yin=np.array([])

yin=y1\*v1+y2\*v2

for i in range(0,4):

if yin[i]>=theta:

y[i]=1

else:

y[i]=0

print("yin",yin)

print('Output of Net')

y=y.astype(int)

print("y",y)

print("z",z)

if np.array\_equal(y,z):

con=0

else:

print("Net is not learning enter another set of weights and Threshold value")

w11=input("Weight w11=")

w12=input("weight w12=")

w21=input("Weight w21=")

w22=input("weight w22=")

v1=input("weight v1=")

v2=input("weight v2=")

theta=input("theta=")

print("McCulloch-Pitts Net for XOR function")

print("Weights of Neuron Z1")

print(w11)

print(w21)

print("weights of Neuron Z2")

print(w12)

print(w22)

print("weights of Neuron Y")

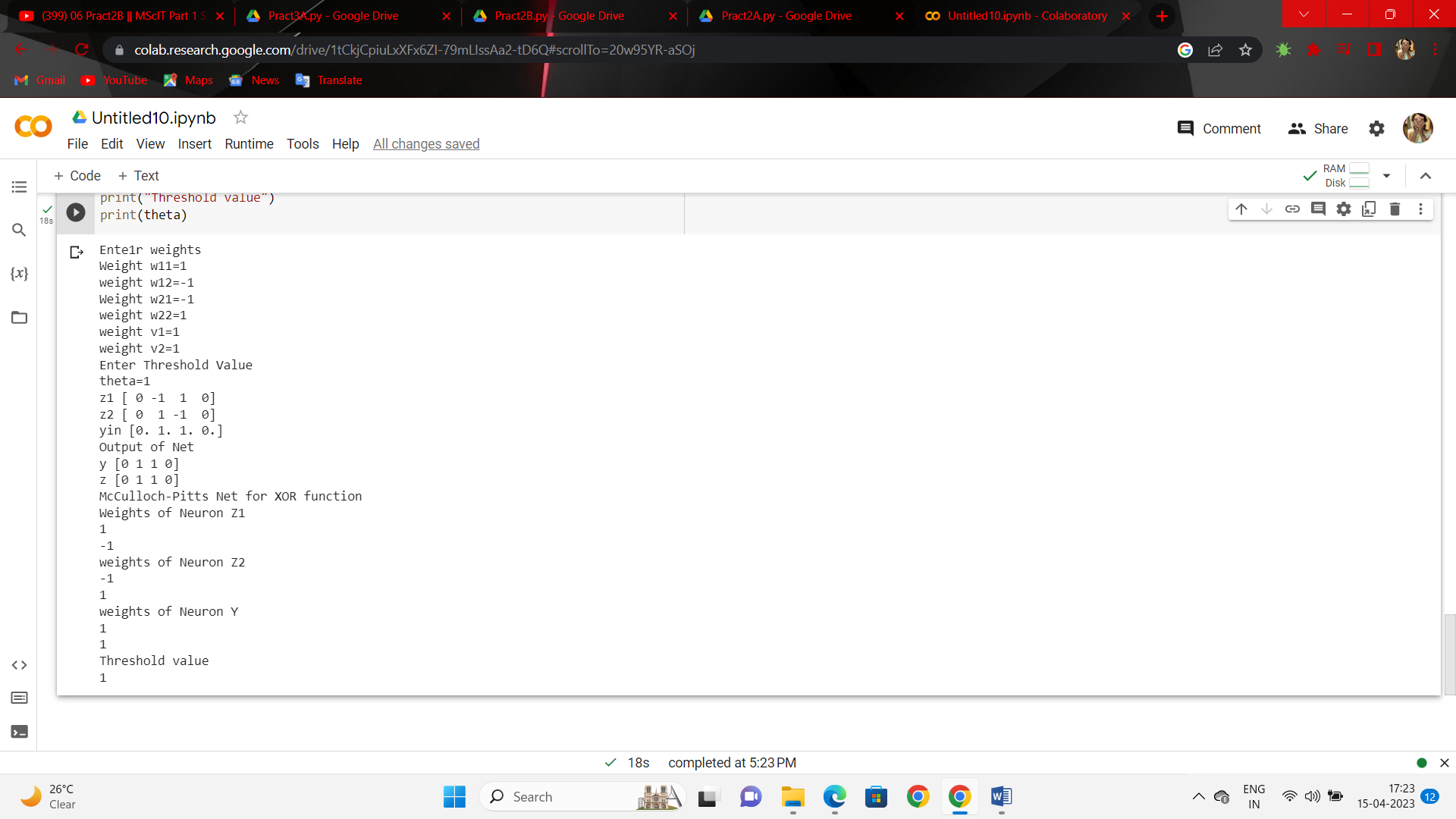
print(v1)

print(v2)

print("Threshold value")

print(theta)

MAKE SURE INPUT VALUES ARE THE SAME SHOWN IN FIGURE OTHERWISE IT GIVES YOU ERROR



PRACT 3A Write a program to implement Hebb’s rule.

CODE:

#Pract 3A

import numpy as np

#first pattern

x1=np.array([1,1,1,-1,1,-1,1,1,1])

#second pattern

x2=np.array([1,1,1,1,-1,1,1,1,1])

#initialize bais value

b=0

#define target

y=np.array([1,-1])

wtold=np.zeros((9,))

wtnew=np.zeros((9,))

wtnew=wtnew.astype(int)

wtold=wtold.astype(int)

bais=0

print("First input with target =1")

for i in range(0,9):

wtold[i]=wtold[i]+x1[i]\*y[0]

wtnew=wtold

b=b+y[0]

print("Second input with target =-1")

for i in range(0,9):

wtnew[i]=wtold[i]+x2[i]\*y[1]

b=b+y[1]

print("new wt =", wtnew)

print("Bias value",b)

PRAC 3B Write a program to implement of delta rule.

CODE:

#Pract 3B

#supervised learning

import numpy as np

import time

np.set\_printoptions(precision=2)

x=np.zeros((3,))

weights=np.zeros((3,))

desired=np.zeros((3,))

actual=np.zeros((3,))

for i in range(0,3):

x[i]=float(input("Initial inputs:"))

for i in range(0,3):

weights[i]=float(input("Initial weights:"))

for i in range(0,3):

desired[i]=float(input("Desired output:"))

a=float(input("Enter learning rate:"))

actual=x\*weights

print("actual",actual)

print("desired",desired)

while True:

if np.array\_equal(desired,actual):

break #no change

else:

for i in range(0,3):

weights[i]=weights[i]+a\*(desired[i]-actual[i])

actual=x\*weights

print("weights",weights)

print("actual",actual)

print("desired",desired)

print("\*"\*30)

print("Final output")

print("Corrected weights",weights)

print("actual",actual)

print("desired",desired)

**NOTE: MAKE SURE ALL INPUTS ARE 1**

**PRACT 4A** Write a program for Back Propagation Algorithm

**CODE:**

#Pract 4A Aim: Implement the back propogation algorithm for the above

# neural network

import numpy as np

import decimal

import math

np.set\_printoptions(precision=2)

v1=np.array([0.6, 0.3])

v2=np.array([-0.1, 0.4])

w=np.array([-0.2,0.4,0.1])

b1=0.3

b2=0.5

x1=0

x2=1

alpha=0.25

print("calculate net input to z1 layer")

zin1=round(b1+ x1\*v1[0]+x2\*v2[0],4)

print("z1=",round(zin1,3))

print("calculate net input to z2 layer")

zin2=round(b2+ x1\*v1[1]+x2\*v2[1],4)

print("z2=",round(zin2,4))

print("Apply activation function to calculate output")

z1=1/(1+math.exp(-zin1))

z1=round(z1,4)

z2=1/(1+math.exp(-zin2))

z2=round(z2,4)

print("z1=",z1)

print("z2=",z2)

print("calculate net input to output layer")

yin=w[0]+z1\*w[1]+z2\*w[2]

print("yin=",yin)

print("calculate net output")

y=1/(1+math.exp(-yin))

print("y=",y)

fyin=y\*(1- y)

dk=(1-y)\*fyin

print("dk=",dk)

dw1= alpha \* dk \* z1

dw2= alpha \* dk \* z2

dw0= alpha \* dk

print("compute error portion in delta")

din1=dk\* w[1]

din2=dk\* w[2]

print("din1=",din1)

print("din2=",din2)

print("error in delta")

fzin1= z1 \*(1-z1)

print("fzin1=",fzin1)

d1=din1\* fzin1

fzin2= z2 \*(1-z2)

print("fzin2=",fzin2)

d2=din2\* fzin2

print("d1=",d1)

print("d2=",d2)

print("Changes in weights between input and hidden layer")

dv11=alpha \* d1 \* x1

print("dv11=",dv11)

dv21=alpha \* d1 \* x2

print("dv21=",dv21)

dv01=alpha \* d1

print("dv01=",dv01)

dv12=alpha \* d2 \* x1

print("dv12=",dv12)

dv22=alpha \* d2 \* x2

print("dv22=",dv22)

dv02=alpha \* d2

print("dv02=",dv02)

print("Final weights of network")

v1[0]=v1[0]+dv11

v1[1]=v1[1]+dv12

print("v1=",v1)

v2[0]=v2[0]+dv21

v2[1]=v2[1]+dv22

print("v2=",v2)

w[1]=w[1]+dw1

w[2]=w[2]+dw2

b1=b1+dv01

b2=b2+dv02

w[0]=w[0]+dw0

print("w=",w)

print("bias b1=",b1, " b2=",b2)

**PRAC 4B** Write a program for error Backpropagation algorithm.

CODE;

import math

a0=-1

t=-1

w10=float(input("Enter weight first network"))

b10=float(input("Enter base first network:"))

w20=float(input("Enter weight second network:"))

b20=float(input("Enter base second network:"))

c=float(input("Enter learning coefficient:"))

n1=float(w10\*c+b10)

a1=math.tanh(n1)

n2=float(w20\*a1+b20)

a2=math.tanh(float(n2))

e=t-a2

s2=-2\*(1-a2\*a2)\*e

s1=(1-a1\*a1)\*w20\*s2

w21=w20-(c\*s2\*a1)

w11=w10-(c\*s1\*a0)

b21=b20-(c\*s2)

b11=b10-(c\*s1)

print("The updated weight of first n/w w11=",w11)

print("The uploaded weight of second n/w w21= ",w21)

print("The updated base of first n/w b10=",b10)

print("The updated base of second n/w b20= ",b20)

**PRACT 7** Write a program for Linear separation

CODE:

import numpy as np

import matplotlib.pyplot as plt

def create\_distance\_function(a, b, c):

""" 0 = ax + by + c """

def distance(x, y):

""" returns tuple (d, pos)

d is the distance

If pos == -1 point is below the line,

0 on the line and +1 if above the line

"""

nom = a \* x + b \* y + c

if nom == 0:

pos = 0

elif (nom < 0 and b < 0) or (nom > 0 and b > 0):

pos = -1

else:

pos = 1

return (np.absolute(nom) / np.sqrt(a \*\* 2 + b \*\* 2), pos)

return distance

points = [(3.5, 1.8), (1.1, 3.9)]

fig, ax = plt.subplots()

ax.set\_xlabel("sweetness")

ax.set\_ylabel("sourness")

ax.set\_xlim([-1, 6])

ax.set\_ylim([-1, 8])

X = np.arange(-0.5, 5, 0.1)

colors = ["r", ""] # for the samples

size = 10

for (index, (x, y)) in enumerate(points):

if index == 0:

ax.plot(x, y, "o",

color="darkorange",

markersize=size)

else:

ax.plot(x, y, "oy",

markersize=size)

step = 0.05

for x in np.arange(0, 1 + step, step):

slope = np.tan(np.arccos(x))

dist4line1 = create\_distance\_function(slope, -1, 0)

# print("x: ", x, "slope: ", slope)

Y = slope \* X

results = []

for point in points:

results.append(dist4line1(\*point))

# print(slope, results)

if (results[0][1] != results[1][1]):

ax.plot(X, Y, "g-")

else:

ax.plot(X, Y, "r-")

plt.show()

**PRACT 8A** Membership and Identity Operators | in, not in,

CODE: list1=[]

c=int(input("Enter the number of elements that you want to insert in List 1:"))

for i in range(0,c):

ele = int(input("Enter the element :"))

list1.append(ele)

a = int(input("enter the number that you want to find in List 1:"))

if a not in list1:

print( "The list does not contain ", a )

else:

print( "The list contains", a )

**PRACT 8B** Membership and Identity Operatorsis, is not

CODE:

my\_new\_list1 = [1, 2, 3, 'a']

my\_new\_list2 = my\_new\_list1

my\_new\_tuple1 = (1, 2, 3, 'a')

my\_new\_tuple2 = (4, 5, 'b', 'a')

print("my\_new\_list1", my\_new\_list1)

print("type of my\_new\_list1",type(my\_new\_list1))

print("my\_new\_list2", my\_new\_list2)

print("type of my\_new\_list2",type(my\_new\_list2))

print("my\_new\_tuple1", my\_new\_tuple1)

print("type of my\_new\_tuple1", type(my\_new\_tuple1))

print("my\_new\_tuple2", my\_new\_tuple2)

print("type of my\_new\_tuple2", type(my\_new\_tuple2))

if type(my\_new\_list1) is not type(my\_new\_tuple1):

print('true!!! ,my\_new\_list1 & my\_new\_tuple1 does not belong to same type')

else:

print("my\_new\_list1 & my\_new\_tuple1 belong to same type")

if type(my\_new\_tuple1) is not type(my\_new\_tuple2):

print('true!!!, my\_new\_tuple1 & my\_new\_tuple2 does not belong to same type')

else:

print("my\_new\_tuple1 & my\_new\_tuple2 belong to same type")

if (my\_new\_list1) is not (my\_new\_list2):

print("true!!!, my\_new\_list1 & my\_new\_list2 are not same")

else:

print("my\_new\_list1 & my\_new\_list2 are same")

**PRACT 9A** Find ratios using fuzzy logic

**CODE1: pip install fuzzywuzzy**

Run above installation add new code and then copy paste the below code

**Code2:** pip install python-Levenshtein

Run the above and add new code and copy paste below then finally its ready to run your actual program

**CODE3:**

# Python code showing all the ratios together,

# make sure you have installed fuzzywuzzy module

from fuzzywuzzy import fuzz

from fuzzywuzzy import process

s1 = "I love GeeksforGeeks"

s2 = "I am loving GeeksforGeeks"

print("FuzzyWuzzy Ratio: ", fuzz.ratio( s1, s2 ))

print("FuzzyWuzzy PartialRatio: ", fuzz.partial\_ratio( s1, s2 ))

print("FuzzyWuzzy TokenSortRatio: ", fuzz.token\_sort\_ratio( s1, s2 ))

print("FuzzyWuzzy TokenSetRatio: ", fuzz.token\_set\_ratio( s1, s2 ))

print("FuzzyWuzzy WRatio: ", fuzz.WRatio( s1, s2 ), '\n\n')

# for process library,

query = 'geeks for geeks'

choices = ['geek for geek', 'geek geek', 'g. for geeks']

print("List of ratios: ")

print(process.extract( query, choices ), '\n')

print("Best among the above list: ", process.extractOne( query, choices ))

**PRACT 9B** Solve Tipping problem using fuzzy logic

CODE1: pip install -U scikit-fuzzy

Code2:

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

import matplotlib.pyplot as plt

# New Antecedent/Consequent objects hold universe variables and membership

# functions

#np.arange will give a array starting from zero to 11, with increment of 1

quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')

service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')

tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')

# Auto-membership function population is possible with .automf(3, 5, or 7)

quality.automf(3)

service.automf(3)

# Custom membership functions can be built interactively with a familiar,

# Pythonic API

#trimf() is triangular membership generator

tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])

tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])

tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])

# You can see how these look with .view()

quality['average'].view()

plt.show(block=True)

service['poor'].view()

plt.show(block=True)

tip['high'].view()

plt.show(block=True)

#rule to implemented

rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])

rule2 = ctrl.Rule(service['average'], tip['medium'])

rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])

rule1.view()

plt.show(block=True)

# rule2.view()

# rule3.view()

tipping\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API

# Note: if you like passing many inputs all at once, use .inputs(dict\_of\_data)

tipping.input['quality'] = 6.5

tipping.input['service'] = 9.8

# Crunch the numbers

tipping.compute()

print (tipping.output['tip'])

tip.view(sim=tipping)

quality['average'].view()

plt.show(block=True)

**pract10A** Implementation of Simple genetic algorithm

CODE:

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

QRSTUVWXYZ 1234567890, .-;:\_!"#%&/()=?@${[]}'''

# Target string to be generated

TARGET = "I love GeeksforGeeks"

class Individual( object ):

'''

Class representing individual in population

'''

def \_\_init\_\_(self, chromosome):

self.chromosome = chromosome

self.fitness = self.cal\_fitness()

@classmethod

def mutated\_genes(self):

'''

create random genes for mutation

'''

global GENES

gene = random.choice( GENES )

return gene

@classmethod

def create\_gnome(self):

'''

create chromosome or string of genes

'''

global TARGET

gnome\_len = len( TARGET )

return [self.mutated\_genes() for \_ in range( gnome\_len )]

def mate(self, par2):

'''

Perform mating and produce new offspring

'''

# chromosome for offspring

child\_chromosome = []

for gp1, gp2 in zip( self.chromosome, par2.chromosome ):

# random probability

prob = random.random()

# if prob is less than 0.45, insert gene

# from parent 1

if prob < 0.45:

child\_chromosome.append( gp1 )

# if prob is between 0.45 and 0.90, insert

# gene from parent 2

elif prob < 0.90:

child\_chromosome.append( gp2 )

# otherwise insert random gene(mutate),

# for maintaining diversity

else:

child\_chromosome.append( self.mutated\_genes() )

# create new Individual(offspring) using

# generated chromosome for offspring

return Individual( child\_chromosome )

def cal\_fitness(self):

'''

Calculate fittness score, it is the number of

characters in string which differ from target

string.

'''

global TARGET

fitness = 0

for gs, gt in zip( self.chromosome, TARGET ):

if gs != gt: fitness += 1

return fitness

# Driver code

def main():

global POPULATION\_SIZE

# current generation

generation = 1

found = False

population = []

# create initial population

for \_ in range( POPULATION\_SIZE ):

gnome = Individual.create\_gnome()

population.append( Individual( gnome ) )

while not found:

# sort the population in increasing order of fitness score

population = sorted( population, key=lambda x: x.fitness )

# if the individual having lowest fitness score ie.

# 0 then we know that we have reached to the target

# and break the loop

if population[0].fitness <= 0:

found = True

break

# Otherwise generate new offsprings for new generation

new\_generation = []

# Perform Elitism, that mean 10% of fittest population

# goes to the next generation

s = int( (10 \* POPULATION\_SIZE) / 100 )

new\_generation.extend( population[:s] )

# From 50% of fittest population, Individuals

# will mate to produce offspring

s = int( (90 \* POPULATION\_SIZE) / 100 )

for \_ in range( s ):

parent1 = random.choice( population[:50] )

parent2 = random.choice( population[:50] )

child = parent1.mate( parent2 )

new\_generation.append( child )

population = new\_generation

print( "Generation: {}\tString: {}\tFitness: {}". \

format( generation,

"".join( population[0].chromosome ),

population[0].fitness ) )

generation += 1

print( "Generation: {}\tString: {}\tFitness: {}". \

format( generation,

"".join( population[0].chromosome ),

population[0].fitness ) )

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

main()

**PRACT 10B** Create two classes: City and Fitness using Genetic algorithm

CODE:

# PRACTICAL NO: 10 (B)

# Title: Solve Travelling salesman problem (TSP) using GA

# Source Code:

# #Aim: Solve Travelling salesman problem (TSP) using GA

import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt

# We first create a City class that will allow us to create and handle our cities.

# These are simply our (x, y) coordinates. Within the City class,

# we add a distance calculation (making use of the Pythagorean theorem)

# and a cleaner way to output the cities as coordinates with \_\_repr\_\_.

class City:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def distance(self, city):

xDis = abs(self.x - city.x)

yDis = abs(self.y - city.y)

distance = np.sqrt((xDis \*\* 2) + (yDis \*\* 2))

return distance

def \_\_repr\_\_(self):

return "(" + str(self.x) + "," + str(self.y) + ")"

# We’ll also create a Fitness class. In our case, we’ll treat the fitness as the inverse of the #route distance.

# We want to minimize route distance, so a larger fitness score is better.

# Based on Rule #2, we need to start and end at the same place,

# so this extra calculation is accounted for the distance calculation.

class Fitness:

def \_\_init\_\_(self, route):

self.route = route

self.distance = 0

self.fitness = 0.0

def routeDistance(self):

if self.distance == 0:

pathDistance = 0

for i in range(0, len(self.route)):

fromCity = self.route[i]

toCity = None

if i + 1 < len(self.route):

toCity = self.route[i + 1]

else:

toCity = self.route[0]

pathDistance += fromCity.distance(toCity)

self.distance = pathDistance

return self.distance

def routeFitness(self):

if self.fitness == 0:

self.fitness = 1 / float(self.routeDistance())

return self.fitness

#Creating the population:

#We now can make our initial population (aka first generation).

# To do so, we need a way to create a function that produces routes that satisfy our #conditions

# To create an individual, we randomly select the order in which we visit each city:

def createRoute(cityList):

route = random.sample(cityList, len(cityList))

return route

#This produces one individual, but we want a full population, so let’s do that in our next #function.

# This is as simple as looping through the createRoute function until

# we have as many routes as we want for our population.

def initialPopulation(popSize, cityList):

population = []

for i in range(0, popSize):

population.append(createRoute(cityList))

return population

#Determine Fitness:

#To simulate our “survival of the fittest”,

# we can make use of Fitness to rank each individual in the population.

# Our output will be an ordered list with the route IDs and each associated fitness score.

def rankRoutes(population):

fitnessResults = {}

for i in range(0,len(population)):

fitnessResults[i] = Fitness(population[i]).routeFitness()

return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)

#Select the mating Pool:

#For the purpose of clarity, we’ll create the mating pool in two steps.

# First, we’ll use the output from rankRoutes to determine which routes to select in our #selection function.

# We set up the roulette wheel by calculating a relative fitness weight for each individual.

# We compare a randomly drawn number to these weights to select our mating pool.

# We’ll also want to hold on to our best routes, so we introduce elitism.

# Ultimately, the selection function returns a list of route IDs,

# which we can use to create the mating pool in the matingPool function.

def selection(popRanked, eliteSize):

selectionResults = []

df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])

df['cum\_sum'] = df.Fitness.cumsum()

df['cum\_perc'] = 100 \* df.cum\_sum / df.Fitness.sum()

for i in range(0, eliteSize):

selectionResults.append(popRanked[i][0])

for i in range(0, len(popRanked) - eliteSize):

pick = 100 \* random.random()

for i in range(0, len(popRanked)):

if pick <= df.iat[i, 3]:

selectionResults.append(popRanked[i][0])

break

return selectionResults

#Now that we have the IDs of the routes that will make up our mating pool from the #selection function,

# we can create the mating pool. We’re simply extracting the selected individuals from our #population.

def matingPool(population, selectionResults):

matingpool = []

for i in range(0, len(selectionResults)):

index = selectionResults[i]

matingpool.append(population[index])

return matingpool

#Breed:

#With our mating pool created, we can create the next generation in a process called crossover (aka “breeding”).

#the TSP is unique in that we need to include all locations exactly one time.

# To abide by this rule, we can use a special breeding function called ordered crossover.

# In ordered crossover, we randomly select a subset of the first parent string (see line 12 in #breed function below)

# and then fill the remainder of the route with the genes from the second parent in the #order in which they appear,

# without duplicating any genes in the selected subset from the first parent

def breed(parent1, parent2):

child = []

childP1 = []

childP2 = []

geneA = int(random.random() \* len(parent1))

geneB = int(random.random() \* len(parent1))

startGene = min(geneA, geneB)

endGene = max(geneA, geneB)

for i in range(startGene, endGene):

childP1.append(parent1[i])

childP2 = [item for item in parent2 if item not in childP1]

child = childP1 + childP2

return child

#we’ll generalize this to create our offspring population.

# We use elitism to retain the best routes from the current population.

# Then, we use the breed function to fill out the rest of the next generation.

def breedPopulation(matingpool, eliteSize):

children = []

length = len(matingpool) - eliteSize

pool = random.sample(matingpool, len(matingpool))

for i in range(0, eliteSize):

children.append(matingpool[i])

for i in range(0, length):

child = breed(pool[i], pool[len(matingpool) - i - 1])

children.append(child)

return children

#Mutate:

#Mutation serves an important function in GA, as it helps to avoid local convergence by #introducing novel routes

# that will allow us to explore other parts of the solution space.

# Similar to crossover, the TSP has a special consideration when it comes to mutation.

#since we need to abide by our rules, we can’t drop cities. Instead, we’ll use swap mutation.

# This means that, with specified low probability, two cities will swap places in our route.

# We’ll do this for one individual in our mutate function:

def mutate(individual, mutationRate):

for swapped in range(len(individual)):

if (random.random() < mutationRate):

swapWith = int(random.random() \* len(individual))

city1 = individual[swapped]

city2 = individual[swapWith]

individual[swapped] = city2

individual[swapWith] = city1

return individual

#Next, we can extend the mutate function to run through the new population.

def mutatePopulation(population, mutationRate):

mutatedPop = []

for ind in range(0, len(population)):

mutatedInd = mutate(population[ind], mutationRate)

mutatedPop.append(mutatedInd)

return mutatedPop

#We’re almost there. Let’s pull these pieces together to create a function that produces a #new generation.

# First, we rank the routes in the current generation using rankRoutes.

# We then determine our potential parents by running the selection function,

# which allows us to create the mating pool using the matingPool function.

# Finally, we then create our new generation using the breedPopulation function and

# then applying mutation using the mutatePopulation function.

def nextGeneration(currentGen, eliteSize, mutationRate):

popRanked = rankRoutes(currentGen)

selectionResults = selection(popRanked, eliteSize)

matingpool = matingPool(currentGen, selectionResults)

children = breedPopulation(matingpool, eliteSize)

nextGeneration = mutatePopulation(children, mutationRate)

return nextGeneration

#We finally have all the pieces in place to create our GA!

# All we need to do is create the initial population, and then we can loop through as many #generations as we desire.

# Of course we also want to see the best route and how much we’ve improved, so we #capture the initial distance

# the final distance and the best route

def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):

pop = initialPopulation(popSize, population)

print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))

for i in range(0, generations):

pop = nextGeneration(pop, eliteSize, mutationRate)

print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))

bestRouteIndex = rankRoutes(pop)[0][0]

bestRoute = pop[bestRouteIndex]

return bestRoute

#Running the genetic algorithm

#First, we need a list of cities to travel between. For this demonstration, we’ll create a list #of 25 random cities

cityList = []

for i in range(0,25):

cityList.append(City(x=int(random.random() \* 200), y=int(random.random() \* 200)))

#Then, running the genetic algorithm is one simple line of code. This is where art meets #science;

# you should see which assumptions work best for you. In this example, we have 100 #individuals in each generation,

# keep 20 elite individuals, use a 1% mutation rate for a given gene, and run through 500 generations:

geneticAlgorithm(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)

# we can store the shortest distance from each generation in a progress list and then plot #the results.

def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):

pop = initialPopulation(popSize, population)

progress = []

progress.append(1 / rankRoutes(pop)[0][1])

for i in range(0, generations):

pop = nextGeneration(pop, eliteSize, mutationRate)

progress.append(1 / rankRoutes(pop)[0][1])

plt.plot(progress)

plt.ylabel('Distance')

plt.xlabel('Generation')

plt.show()

#Run the GA in the same way as before, but now using the newly created #geneticAlgorithmPlot function

geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)