**INDEX**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Prac.**  **No** |  | **Practical** | **Pg.No** | **Date** | **Sign** |
| 1. |  | Implement the following |  |  |  |
|  | A | Design a simple Linear neural network model. | 3 |  |  |
|  | B | Calculate the output of neural net using both binary and Bipolar sigmoidal function. | 4 |  |  |
| 2. |  | Implement the following |  |  |  |
|  | A | Generate And/Not function using McCulloch Pitts neural net. | 6 |  |  |
|  | B | Generate XOR function using McCulloch Pitts neural net. | 8 |  |  |
| 3. |  | Implement the following |  |  |  |
|  | A | Write a program to implement Hebb’s rule. | 11 |  |  |
|  | B | Write a program to implement Delta rule. | 12 |  |  |
| 4. |  | Implement the following |  |  |  |
|  | A | Write a program for basic back propogation algorithm. | 15 |  |  |
|  | B | Write a program for error back propogation algorithm. | 16 |  |  |
| 5. |  | Implement the following |  |  |  |
|  | A | Write a program for Hopfield network. | 19 |  |  |
|  | B | Write a program for radial basis function. | 20 |  |  |
| 6. |  | Implement the following |  |  |  |
|  | A | Kohonen self-organizing map. | 23 |  |  |
|  | B | Adaptive Resonance theory. | 24 |  |  |
| 7. |  | Implement the following |  |  |  |
|  | A | Write a program for linear  separation. | 28 |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | B | Write a program for Hopfield network model for associative memory | 30 |  |  |
| 8. |  | Implement the following |  |  |  |
|  | A. | Membership and Identity operators (in/not in) | 34 |  |  |
|  | B. | Membership and Identity operators (is/is not) | 36 |  |  |
| 9. |  | Implement the following |  |  |  |
|  | A | Find ratios using fuzzy logic | 38 |  |  |
|  | B | Solve Tipping problem using fuzzy logic | 39 |  |  |
| 10. |  | Implement the following |  |  |  |
|  | A | Implementation of simple genetic algorithm. | 42 |  |  |
|  | B | Create two classes: City and Fitness using a Genetic Algorithm. | 46 |  |  |

Practical No: 1

Theory:

Neural networks are artificial systems that were inspired by biological neural networks. These systems learn to perform tasks by being exposed to various datasets and examples without any task-specific rules. Neural networks are based on computational models for threshold logic. Threshold logic is a combination of algorithms and mathematics. Neural networks are based either on the study of the brain or on the application of neural networks to artificial intelligence. The work has led to improvements in finite automata theory. Components of a typical neural network involve neurons, connections which are known as synapses, weights, biases, propagation function, and a learning rule.

A] Design a simple linear neural network model.

Calculate the output of neural net where input X = 0.3, w = 0.2 and bias b 0.4.

Given neural net:

Yin = wx + b = 0.2\*0.3 + 0.4 = 0.46.

if (yin &lt; 0), then output=y=0

else if (yin &gt;1) then output=y=1

else output=y=yin

Code:

x = float(input("Entre the value of x"))

b = float(input("Entre the value of bias"))

w = float(input("Entre the value of weight"))

print("x=" ,x)

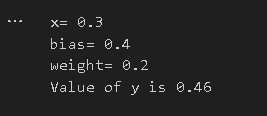
print("bias=" ,b)

print("weight=" ,w)

y = w\*x + b

print(f'Value of y is {y}')

output:-



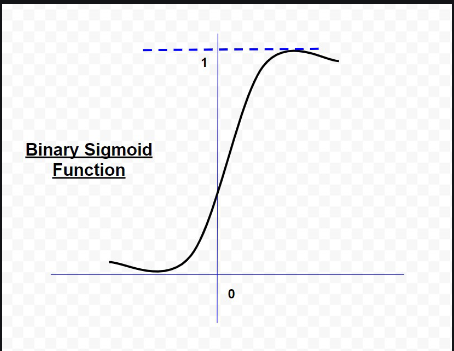
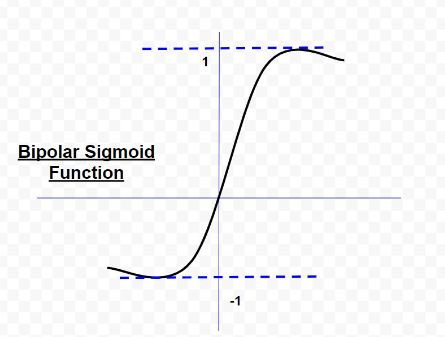
B] Calculate the output of neural net using both binary and bipolar sigmoidal function.

**Theory:**

**- Binary Sigmoid function** is by far the most commonly used activation function in neural networks. The need for a sigmoid function stems from the fact that many learning algorithms require the activation function to be differentiable and hence continuous.

A binary sigmoid function is of the form: 

where k = steepness or slope parameter, By varying the value of k, a sigmoid function with different slopes can be obtained. It has a range of (0,1). The slope of origin is k/4. As the value of k becomes very large, the sigmoid function becomes a threshold function.

- A **bipolar sigmoid function** is of the form : 

The range of values of sigmoid functions can be varied depending on the application. However, the range of (-1,+1) is most commonly adopted.

#Sigmoid and Bipolar

import numpy as np

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

w = []

x = []

#taking the values of input and their weight

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 = 0.0

for i in range(0,n):

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

print("The net input y is")

print(y)

#Applying Binary Sigmoid fuction on the net input i.e y

binary = 1/(1+np.e\*\*(-y))

print ("Binary output after applying sigmoid function:")

print(round(binary,3))

#Applying Bipolor Sigmoid fuction on the net input i.e y

bipolar = -1+(2/(1+(np.e\*\*(-y))))

print("The output after applying Bipolar function:")

print(round(bipolar,3))

Output:-

A screenshot of a computer program

Description automatically generated

Practical No: 2

**Theory**:

The McCulloch-Pitts neural model, which was the earliest ANN model, has only two types of inputs — Excitatory and Inhibitory. The excitatory inputs have weights of positive magnitude and the inhibitory weights have weights of negative magnitude. The inputs of the McCulloch-Pitts neuron could be either 0 or 1. It has a threshold function as an activation function. So, the output signal yout is 1 if the input ysum is greater than or equal to a given threshold value, else 0.

A] Generate AND/NOT function using McCulloch-Pitts neural net.

Code:

threshold=0.5

x1=[]

x2=[]

w1 = 1

w2 =-1

yt=[]

yout=[]

for i in range(4):

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

x1.append(a)

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

x2.append(b)

c=int(input("Enter thr input yt: "))

yt.append(c)

print("x1=",x1)

print("x2=",x2)

print("yt=",yt)

for j in range(4):

y=x1[j]\*w1 + x2[j]\*w2

if(y>threshold):

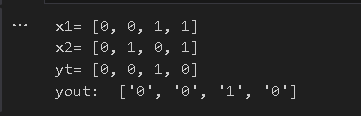
yout.append('1')

else:

yout.append('0')

print("yout: ", yout)

Output:-



B] Generate XOR function using McCulloch-Pitts neural net.

Code:

theta = 1

def output(w,X):

return [int(x1\*w[0] + x2\*w[1] >=1) for x1,x2 in X]

X = [(0,0),(0,1),(1,0),(1,1)]

print(f'X = {X}')

w1 = [1,-1]

Y1 = output(w1,X)

print(f'Y1 = {Y1}')

w2 = [-1,1]

Y2 = output(w2,X)

print(f'Y2 = {Y2}')

Y = list(zip(Y1,Y2))

print(f'Y = {Y}')

w3 = [1,1]

Z = output(w3,Y)

print(f'Z = {Z}')

Output:-

A black screen with white numbers

Description automatically generated

Practical No: 3

A] Write a program to implement Hebb’s rule.

**Theory:**

Hebb’s rule is a postulate proposed by Donald Hebb in 1949. It is a learning rule that describes how neuronal activities influence the connection between neurons, i.e., synaptic plasticity. It provides an algorithm to update the weight of neuronal connections within the neural networks. Hebb’s rule provides a simplistic physiology-based model to mimic the activity-dependent features of synaptic plasticity and has been widely used in the area of artificial neural network.

Code:

import numpy as np

#fisrt 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])

wold = np.zeros((9,))

wnew = np.zeros((9,))

wnew=wnew.astype(int)

bais=0

print("First input with target = 1")

for i in range(0,9):

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

wnew=wold

b=b+y[0]

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

for i in range(0,9):

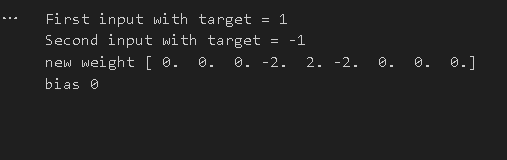
wnew[i]=wold[i]+x2[i]\*y[1]

b=b+y[1]

print("new weight", wnew)

print("bias", b)

Output:



B] Write a program to implement Delta rule

**Theory:**

The Delta rule in machine learning and neural network environments is a specific type of backpropagation that helps to refine connectionist ML/AI networks, making connections between inputs and outputs with layers of artificial neurons.

The Delta rule is also known as the Delta learning rule. Backpropagation has to do with recalculating input weights for artificial neurons using a gradient method. Delta learning does this using the difference between a target activation and an actual obtained activation. Using a linear activation function, network connections are adjusted. Another way to explain the Delta rule is that it uses an error function to perform gradient descent learning.

Code:

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("Initial Desired:"))

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

print("Actual",actual)

print("Desired",desired)

while True:

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

break

else:

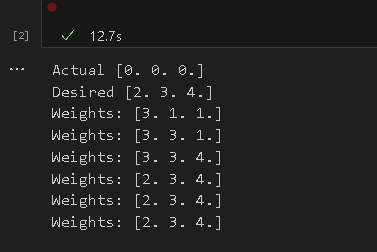
for i in range(0,3):

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

actual = x\*weights

print("Weights:",weights)

Output:



Practical No:4

A] Write a program for basic back propogation algorithm.

**Theory:**

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the error rate obtained in the previous epoch (i.e., iteration). Proper tuning of the weights allows you to reduce error rates and make the model reliable by increasing its generalization. Backpropagation in neural network is a short form for “backward propagation of errors.” It is a standard method of training artificial neural networks. This method helps calculate the gradient of a loss function with respect to all the weights in the network.

**Code:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

#scale units

X=X/np.amax(X,axis=0)

Y=Y/100;

class NN(object):

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

self.inputsize=2

self.outputsize=1

self.hiddensize=3

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize)

def forward(self,X):

self.z=np.dot(X,self.W1)

self.z2=self.sigmoidal(self.z)

self.z3=np.dot(self.z2,self.W2)

op=self.sigmoidal(self.z3)

return op;

def sigmoidal(self,s):

return 1/(1+np.exp(-s))

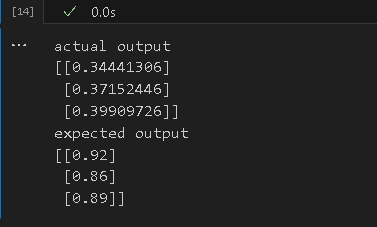
obj=NN()

op=obj.forward(X)

print("actual output\n"+str(op))

print("expected output\n"+str(Y))

**Output:**



B] Write a program for Error BackPropagation Algorithm.

**Theory:**

Error backpropagation. For hidden units, we must propagate the error back from the output nodes (hence the name of the algorithm). Again using the chain rule, we can expand the error of a hidden unit in terms of its posterior nodes:

**Code:**

import numpy as np

X=np.array(([2,9],[1,5],[3,6]),dtype=float)

Y=np.array(([92],[86],[89]),dtype=float)

X=X/np.amax(X,axis=0)

Y=Y/100;

class NN(object):

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

self.inputsize=2

self.outputsize=1

self.hiddensize=3

self.W1=np.random.randn(self.inputsize,self.hiddensize)

self.W2=np.random.randn(self.hiddensize,self.outputsize)

def forward(self,X):

self.z=np.dot(X,self.W1)

self.z2=self.sigmoidal(self.z)

self.z3=np.dot(self.z2,self.W2)

op=self.sigmoidal(self.z3)

return op;

def sigmoidal(self,s):

return 1/(1+np.exp(-s))

def sigmoidalprime(self,s):

return s\* (1-s)

def backward(self,X,Y,o):

self.o\_error=Y-o

self.o\_delta=self.o\_error \* self.sigmoidalprime(o)

self.z2\_error=self.o\_delta.dot(self.W2.T)

self.z2\_delta=self.z2\_error \* self.sigmoidalprime(self.z2)

self.W1 = self.W1 + X.T.dot(self.z2\_delta)

self.W2= self.W2+ self.z2.T.dot(self.o\_delta)

def train(self,X,Y):

o=self.forward(X)

self.backward(X,Y,o)

obj=NN()

for i in range(2000):

print("input"+str(X))

print("Actual output"+str(Y))

print("Predicted output"+str(obj.forward(X)))

print("loss"+str(np.mean(np.square(Y-obj.forward(X)))))

obj.train(X,Y)

Output:

A screenshot of a computer

Description automatically generated

Practical No:5

A] Write a program for Hopfield Network.

**Theory:**

• A Hopfield network is a single-layered and recurrent network in which the neurons are entirely connected, i.e., each neuron is associated with other neurons. If there are two neurons i and j, then there is a connectivity weight wij lies between them which is symmetric wij = wji .

• A Hopfield network is at first prepared to store various patterns or memories. Afterward, it is ready to recognize any of the learned patterns by uncovering partial or even some corrupted data about that pattern, i.e., it eventually settles down and restores the closest pattern. Thus, similar to the human brain, the Hopfield model has stability in pattern recognition.

• There are two different approaches to update the nodes:

1] Synchronously: In this approach, the update of all the nodes taking place simultaneously at each time.

2] Asynchronously: In this approach, at each point of time, update one node chosen randomly or according to some rule. Asynchronous updating is more biologically realistic.

**Code:**

import numpy as np

def compute\_next\_state(state,weight):

next\_state = np.where(weight @ state>= 0, +1, -1)

return next\_state

def compute\_final\_state(initial\_state,weight,max\_iter=1000):

previous\_state = initial\_state

next\_state =compute\_next\_state(previous\_state,weight)

is\_stable = np.all(previous\_state == next\_state)

n\_iter = 0

while(not is\_stable) and (n\_iter <= max\_iter):

previous\_state = next\_state;

next\_state = compute\_next\_state(previous\_state,weight)

is\_stable = np.all(previous\_state==next\_state)

n\_iter +=1

return previous\_state, is\_stable,n\_iter

initial\_state = np.array([+1,-1,-1,-1])

weight = np.array([

[0, -1, -1, +1],

[-1, 0, +1, -1],

[-1,+1, 0, -1],

[+1,-1, -1, 0]])

final\_state, is\_stable, n\_iter = compute\_final\_state(initial\_state,weight)

print("Final state",final\_state)

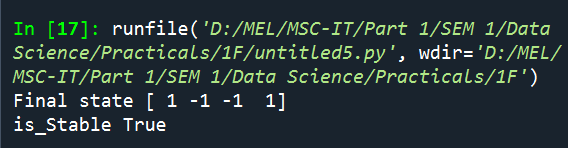
print("is\_Stable",is\_stable)

Output:

A screenshot of a computer

Description automatically generated

B] Write a program for Radial Basis function.



## **B:Write a program for Radial Basis function .**

**Theory:**

• Radial Basis Function (RBF) Networks are a particular type of Artificial Neural Network used for function approximation problems. RBF Networks differ from other neural networks in their three-layer architecture, universal approximation, and faster learning speed.

•Radial Basis Functions are a special class of feed-forward neural networks consisting of three layers: an input layer, a hidden layer, and the output layer. The input layer receives input data and passes it into the hidden layer, where the computation occurs. The hidden layer of Radial Basis Functions Neural Network is the most powerful and very different from most Neural networks. The output layer is designated for prediction tasks like classification or regression.

• Advantages of RBFN

- Easy Design

- Good Generalization

**Code:** R Compiler

D <- matrix(c(-3,1,4), ncol=1)

N <- length(D)

rbf.gauss <- function(gamma=1.0) {

function(x){

exp(-gamma \* norm(as.matrix(x),"F")^2)

}

}

xlim <- c(-5,7)

print(N)

print(xlim)

plot(NULL,xlim=xlim,ylim=c(0,1.25), type = "n")

points(D,rep(0,length(D)), col= 1:N,pch=19)

x.coord = seq(-7,7,length=250)

gamma <- 1.5

for (i in 1:N){

points(x.coord, lapply(x.coord - D[i,],rbf.gauss(gamma)),type="l",col=i)

}

Output:

**A white background with black text

Description automatically generated**

**A graph of a function

Description automatically generated**

Practical No:6

A] Kohonen Self organizing map.

**Theory:**

Self Organizing Map (or Kohonen Map or SOM) is a type of Artificial Neural Network which is also inspired by biological models of neural systems from the 1970s. It follows an unsupervised learning approach and trained its network through a competitive learning algorithm. SOM is used for clustering and mapping (or dimensionality reduction) techniques to map multidimensional data onto lower-dimensional which allows people to reduce complex problems for easy interpretation. SOM has two layers, one is the Input layer and the other one is the Output layer.

The architecture of the Self Organizing Map with two clusters and n input features of any sample is given below:

**Code:**

import minisom

from minisom import MiniSom

import matplotlib.pyplot as plt

data = [[ 0.80, 0.55, 0.22, 0.03],

[ 0.82, 0.50, 0.23, 0.03],

[ 0.80, 0.54, 0.22, 0.03],

[ 0.80, 0.53, 0.26, 0.03],

[ 0.79, 0.56, 0.22, 0.03],

[ 0.75, 0.60, 0.25, 0.03],

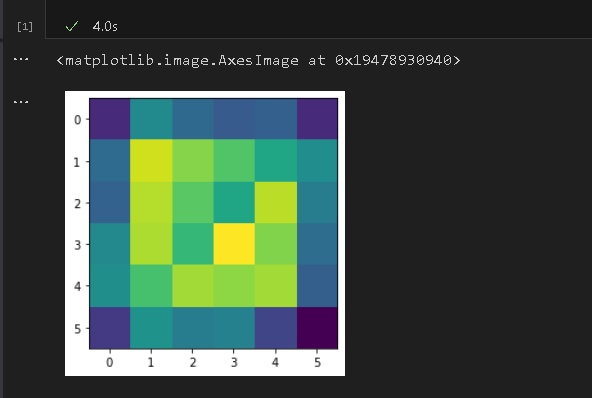
[ 0.77, 0.59, 0.22, 0.03]]

som = MiniSom(6, 6, 4, sigma=0.3, learning\_rate=0.5) # initialization of 6x6 SOM

som.train\_random(data, 100) # trains the SOM with 100 iterations

plt.imshow(som.distance\_map())

Output:



B] Adaptive Resonance Theory.

**Theory:**

Adaptive resonance theory is a type of neural network technique. The basic ART uses unsupervised learning technique. The term “adaptive” and “resonance” used in this suggests that they are open to new learning(i.e. adaptive) without discarding the previous or the old information(i.e. resonance). The ART networks are known to solve the stability-plasticity dilemma i.e., stability refers to their nature of memorizing the learning and plasticity refers to the fact that they are flexible to gain new information. Due to this the nature of ART they are always able to learn new input patterns without forgetting the past. ART networks implement a clustering algorithm.

**Code:**

from \_\_future\_\_ import print\_function

from \_\_future\_\_ import division

import numpy as np

class ART:

def \_\_init\_\_(self, n=5, m=10, rho=.5):

self.F1 = np.ones(n)

self.F2 = np.ones(m)

self.Wf = np.random.random((m,n))

self.Wb = np.random.random((n,m))

self.rho = rho

self.active = 0

def learn(self, X):

self.F2[...] = np.dot(self.Wf, X)

I = np.argsort(self.F2[:self.active].ravel())[::-1]

for i in I:

d = (self.Wb[:,i]\*X).sum()/X.sum()

if d >= self.rho:

self.Wb[:,i] \*= X

self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())

return self.Wb[:,i], i

if self.active < self.F2.size:

i = self.active

self.Wb[:,i] \*= X

self.Wf[i,:] = self.Wb[:,i]/(0.5+self.Wb[:,i].sum())

self.active += 1

return self.Wb[:,i], i

return None,None

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

np.random.seed(1)

network = ART( 5, 10, rho=0.5)

data = [" O ",

" O O",

" O",

" O O",

" O",

" O O",

" O",

" OO O",

" OO ",

" OO O",

" OO ",

"OOO ",

"OO ",

"O ",

"OO ",

"OOO ",

"OOOO ",

"OOOOO",

"O ",

" O ",

" O ",

" O ",

" O",

" O O",

" OO O",

" OO ",

"OOO ",

"OO ",

"OOOO ",

"OOOOO"]

X = np.zeros(len(data[0]))

for i in range(len(data)):

for j in range(len(data[i])):

X[j] = (data[i][j] == 'O')

Z, k = network.learn(X)

print("|%s|"%data[i],"-> class", k)

Output:

A screenshot of a computer program

Description automatically generated

Practical No:7

A] Write a program for Linear Separation.

**Theory:**

Linear separability is the concept wherein the separation of input space into regions is based on whether the network response is positive or negative.

A decision line is drawn to separate positive and negative responses. The decision line may also be called as the decision-making Line or decision-support Line or linear-separable line. The necessity of the linear separability concept was felt to clarify classify the patterns based upon their output responses.

Generally, the net input calculated to the output unit is given as -



⇒ The linear separability of the network is based on the decision-boundary line. If there exist weight for which the training input vectors having a positive (correct) response, or lie on one side of the decision boundary and all the other vectors having negative, −1, response lies on the other side of the decision boundary then we can conclude the problem is "Linearly Separable".

**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("Sweet")

ax.set\_ylabel("Sour")

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()

print('BHavik')

Output:

A screenshot of a graph

Description automatically generated

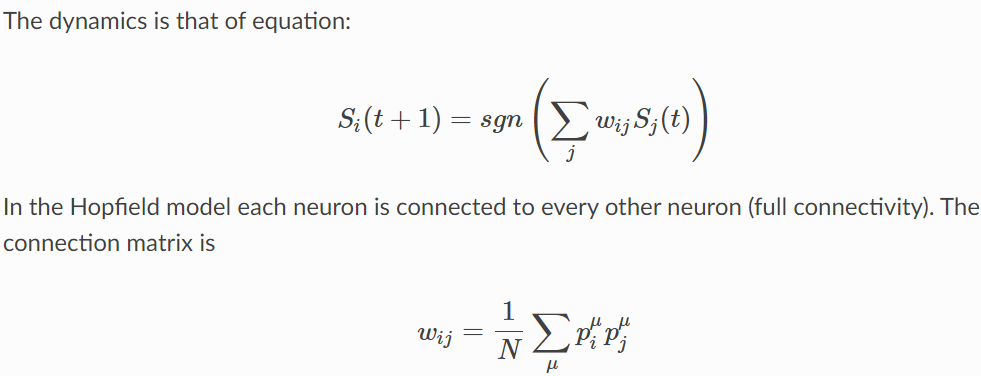
B] Write a program for Hopfield Network model for associative memory.

**Theory:**

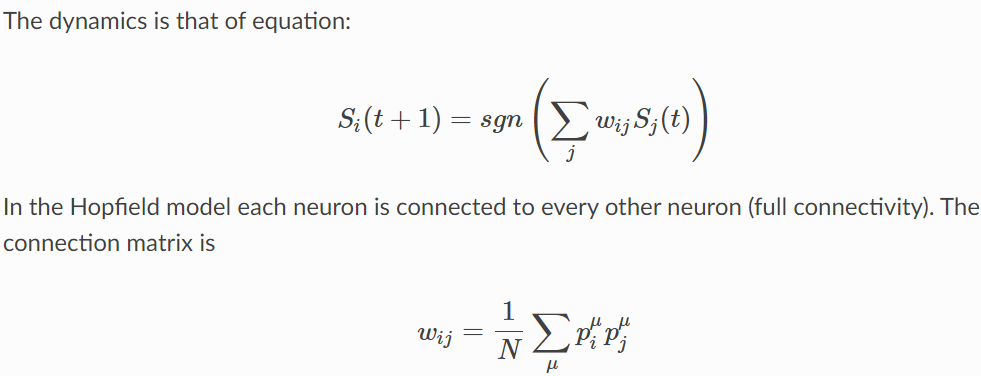
*Hopfield networks* are a special kind of recurrent neural networks that can be used as associative memory. Associative memory is memory that is addressed through its contents. That is, if a pattern is presented to an associative memory, it returns whether this pattern coincides with a stored pattern. An associative memory may also return a stored pattern that is similar to the presented one, so that noisy input can also be recognized.

Hopfield networks are used as associative memory by exploiting the property that they possess stable states, one of which is reached by carrying out the normal computations of a Hopfield network. If the connection weights of the network are determined in such a way that the patterns to be stored become the stable states of the network, a Hopfield network produces for any input pattern a similar stored pattern. Thus noisy patterns can be corrected or distorted patterns can still be recognized.

The dynamics is that of Equation:



In the Hopfield model each neuron is connected to every other neuron. The connection matrix is:



**Code:**

from neurodynex.hopfield\_network import network, pattern\_tools, plot\_tools

import matplotlib.pyplot as plt

pattern\_size = 5

# create an instance of the class HopfieldNetwork

hopfield\_net = network.HopfieldNetwork(nr\_neurons= pattern\_size\*\*2)

# instantiate a pattern factory

factory = pattern\_tools.PatternFactory(pattern\_size, pattern\_size)

# create a checkerboard pattern and add it to the pattern list

checkerboard = factory.create\_checkerboard()

pattern\_list = [checkerboard]

# add random patterns to the list

pattern\_list.extend(factory.create\_random\_pattern\_list(nr\_patterns=3, on\_probability=0.5))

plot\_tools.plot\_pattern\_list(pattern\_list)

# how similar are the random patterns and the checkerboard? Check the overlaps

overlap\_matrix = pattern\_tools.compute\_overlap\_matrix(pattern\_list)

plot\_tools.plot\_overlap\_matrix(overlap\_matrix)

# let the hopfield network "learn" the patterns. Note: they are not stored

# explicitly but only network weights are updated !

hopfield\_net.store\_patterns(pattern\_list)

# create a noisy version of a pattern and use that to initialize the network

noisy\_init\_state = pattern\_tools.flip\_n(checkerboard, nr\_of\_flips=4)

hopfield\_net.set\_state\_from\_pattern(noisy\_init\_state)

# from this initial state, let the network dynamics evolve.

states = hopfield\_net.run\_with\_monitoring(nr\_steps=4)

# each network state is a vector. reshape it to the same shape used to create the patterns.

states\_as\_patterns = factory.reshape\_patterns(states)

# plot the states of the network

plot\_tools.plot\_state\_sequence\_and\_overlap(states\_as\_patterns, pattern\_list,

reference\_idx=0, suptitle="Network dynamics")

print("BHavik")

**output:**



Practical No: 8

A] Membership and Identity Operators | in, not in,

**Theory:**

Python offers two membership operators to check or validate the membership of a value. It tests for membership in a sequence, such as strings, lists, or tuples.

*in operator:* The ‘in’ operator is used to check if a character/ substring/ element exists in a sequence or not. Evaluate to True if it finds the specified element in a sequence otherwise False.

*‘not in’ operato*r- Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Identity operators

Identity operators are used to compare the objects if both the objects are actually of the same data type and share the same memory location.

There are different identity operators such as

‘is’ operator – Evaluates to True if the variables on either side of the operator point to the same object and false otherwise.

‘is not’ operator: Evaluates True if both variables are not the same object.

**Code:**

import skfuzzy as fuzz

import numpy as np

import matplotlib.pyplot as plt

#Creating universe of discourse

x = np.arange(150,191,1) #height of a person

#Creat fuzzy set

short = fuzz.trimf(x,[150,160,170])

medium = fuzz.trimf(x,[160,170,180])

tall = fuzz.trimf(x,[170,180,190])

#Visualize the membership fuction

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

#Plot medium height

plt.plot(x,medium, color='green',label='Medium', linewidth=2)

#Plot tall heights

plt.plot(x,tall, color='red',label='Tall', linewidth=2)

#plot short height

plt.plot(x,short,color='blue', label='Short',linewidth=2)

#Test Data

test\_heights = [153,162,175,182]

#Calculate the membership values for the given height in the tall an medium fuzzy sets

for height in test\_heights:

membership\_tall = fuzz.interp\_membership(x,tall,height)

membership\_medium = fuzz.interp\_membership(x,medium,height)

membership\_short = fuzz.interp\_membership(x,short,height)

print(f'Test Height: {height}')

print(f'Memship in tall: {membership\_tall}')

print(f'Membership in Medium: {membership\_medium}')

print(f'Membership in Short: {membership\_short}\n')

print()

#Highlight the memebership values of test height

for height in test\_heights:

plt.axvline(x=height, color = 'green', linestyle = '--', linewidth = 2)

plt.xlabel('Height')

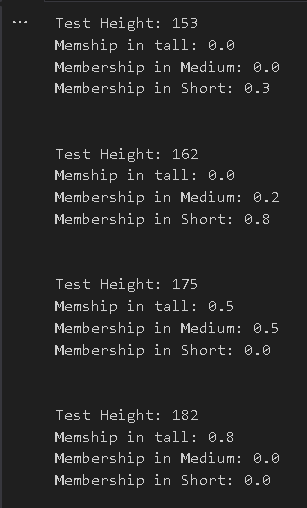
plt.ylabel('Fuzzy Set Membership Values')

plt.legend()

plt.title('Fuzzy set membership')

plt.show()

Output:-



A diagram of a fuzzy set

Description automatically generated

B] Membership and Identity Operators is, is not

Code:

import skfuzzy as fuzz

import numpy as np

import matplotlib.pyplot as plt

# Create a universe of Discourse

x = np.arange(0,11,1)

# Create a fuzzy sets

small = fuzz.trimf(x,[0,0,5])

medium = fuzz.trimf(x,[0,5,10])

# Define test values

test\_value1 = 2 # A value that 'is' in the small set

test\_value2 = 8 # A value that 'is not' in the small set

# Calculate membership values for the test values

membership\_small1 = fuzz.interp\_membership(x,small,test\_value1)

membership\_medium1 = fuzz.interp\_membership(x,medium,test\_value1)

membership\_small2 = fuzz.interp\_membership(x,small,test\_value2)

membership\_medium2 = fuzz.interp\_membership(x,medium,test\_value2)

# Check if a value 'is' or 'is not' in the fuzzy sets

is\_in\_small1 = membership\_small1>0.5

is\_in\_small2 = membership\_small2>0.5

is\_in\_medium1 = membership\_medium1>0.5

is\_in\_medium2 = membership\_medium2>0.5

# Create plots to visualise membership values

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

plt.subplot(121)

plt.plot(x,small,'b',linewidth=2,label='Small')

plt.fill\_between(x,0,small,alpha=0.2)

plt.plot([test\_value1,test\_value1],[0,membership\_small1],'r',linestyle='--',linewidth=2)

plt.text(3,0.8,f'Membership={membership\_small1}',fontsize=12)

plt.title('Membership in "Small" Set')

plt.xlabel('x')

plt.ylabel('Membership')

plt.subplot(122)

plt.plot(x,medium,'b',linewidth=2,label='Medium')

plt.fill\_between(x,0,medium,alpha=0.2)

plt.plot([test\_value1,test\_value1],[0,membership\_medium1],'r',linestyle='--',linewidth=2)

plt.text(3,0.8,f'Membership={membership\_medium1}',fontsize=12)

plt.title('Membership in "Medium" Set')

plt.xlabel('x')

plt.ylabel('Membership')

# Display the plots

plt.tight\_layout()

plt.show()

# Print the results

print(f'Test Value 1: {test\_value1}')

print(f'Membership in small: {membership\_small1}')

print(f'Membership in medium: {membership\_medium1}')

print(f'Is in Small: {is\_in\_small1}\n')

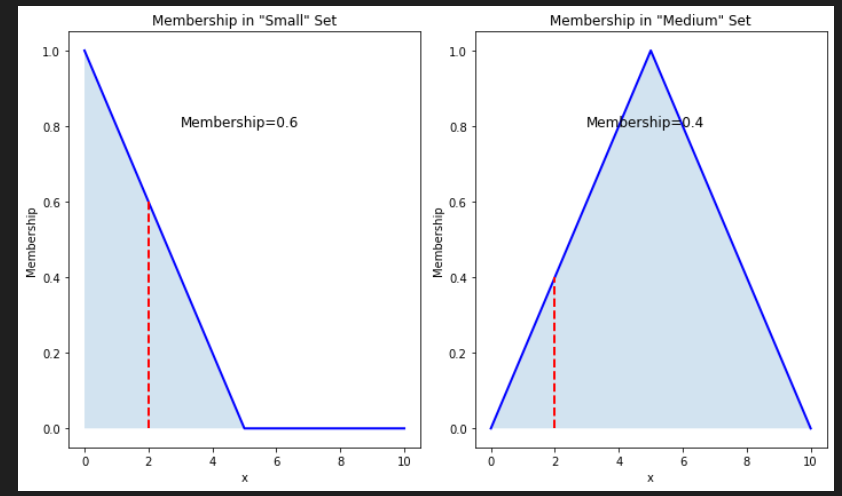
print(f'Test Value 2: {test\_value2}')

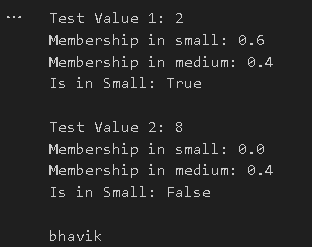
print(f'Membership in small: {membership\_small2}')

print(f'Membership in medium: {membership\_medium2}')

print(f'Is in Small: {is\_in\_small2}\n')

Output: -





Practical No: 9

A] Find ratios using fuzzy logic

Code:

# computing similarity Ratio

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

# Define linguistic variables and their ranges

similarity\_ratio = ctrl.Consequent(np.arange(0, 101, 1), 'Similarity Ratio')

char\_similarity = ctrl.Antecedent(np.arange(0, 101, 1), 'Character Similarity')

# Define linguistic terms and their membership functions

similarity\_ratio['low'] = fuzz.trimf(similarity\_ratio.universe, [0, 0, 50])

similarity\_ratio['medium'] = fuzz.trimf(similarity\_ratio.universe, [0, 50, 100])

similarity\_ratio['high'] = fuzz.trimf(similarity\_ratio.universe, [50, 100, 100])

char\_similarity['low'] = fuzz.trimf(char\_similarity.universe, [0, 0, 50])

char\_similarity['medium'] = fuzz.trimf(char\_similarity.universe, [0, 50, 100])

char\_similarity['high'] = fuzz.trimf(char\_similarity.universe, [50, 100, 100])

# Define fuzzy rules

rule1 = ctrl.Rule(char\_similarity['low'], similarity\_ratio['low'])

rule2 = ctrl.Rule(char\_similarity['medium'], similarity\_ratio['medium'])

rule3 = ctrl.Rule(char\_similarity['high'], similarity\_ratio['high'])

# Create a control system

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

# Create a simulation

similarity\_sim = ctrl.ControlSystemSimulation(similarity\_ctrl)

# Function to compute character similarity

def calculate\_char\_similarity(str1, str2):

common\_chars = set(str1.lower()) & set(str2.lower())

return len(common\_chars) / max(len(str1), len(str2)) \* 100

# Provide input values for the two strings

str\_A = "Gunner William Kline"

str\_B = "Kline, Gunner William"

# Calculate the character similarity

char\_similarity\_value = calculate\_char\_similarity(str\_A, str\_B)

# Compute the similarity ratio using fuzzy logic

similarity\_sim.input['Character Similarity'] = char\_similarity\_value

similarity\_sim.compute()

# Get the computed similarity ratio

similarity\_ratio\_value = similarity\_sim.output['Similarity Ratio']

# Print the similarity ratio

print("Fuzzy Similarity Ratio:", similarity\_ratio\_value)

# Visualize the membership functions and the input

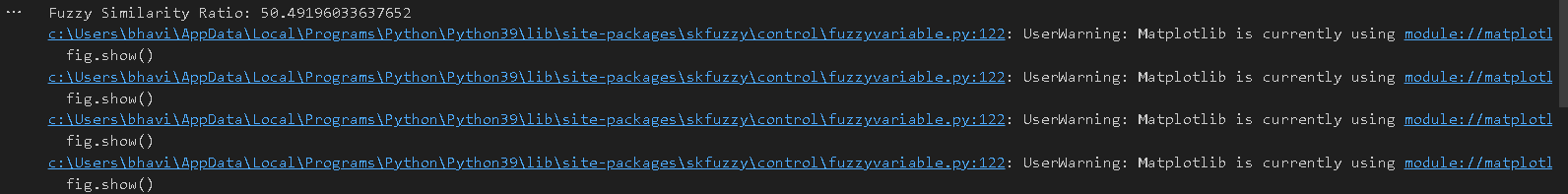
char\_similarity.view()

similarity\_ratio.view()

char\_similarity.view(similarity\_sim)

similarity\_ratio.view(similarity\_sim)

Output:-



A screenshot of a graph

Description automatically generated

A screenshot of a screenshot of a graph

Description automatically generated

B] Solve Tipping problem using fuzzy logic

Code:

import numpy as np

import skfuzzy as fuzz

from skfuzzy import control as ctrl

import matplotlib.pyplot as plt

# Define fuzzy variables and their ranges

service\_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'Service Quality')

food\_quality = ctrl.Antecedent(np.arange(0, 11, 1), 'Food Quality')

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

# Define fuzzy sets and their membership functions

service\_quality.automf(3)

food\_quality.automf(3)

tip.automf(3)

# Define fuzzy rules

rule1 = ctrl.Rule(service\_quality['poor'] | food\_quality['poor'], tip['poor'])

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

rule3 = ctrl.Rule(service\_quality['good'] | food\_quality['good'], tip['good'])

# Create a control system

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

# Create a simulation

tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)

# Provide input values

tipping.input['Service Quality'] = 10

tipping.input['Food Quality'] = 10

# Compute tip

tipping.compute()

# Visualize the membership functions

service\_quality.view()

food\_quality.view()

tip.view()

# Print the result

print("Recommended Tip Amount:", tipping.output['Tip Amount'], "%")

plt.show()

Output:-

A black screen with blue dots

Description automatically generated

A graph with lines and numbers

Description automatically generated with medium confidence

A screenshot of a graph

Description automatically generated

Practical No: 10

A] Implementation of Simple genetic algorithm

**Theory:**

The genetic algorithm is a method for solving both constrained and unconstrained optimization problems that is based on natural selection, the process that drives biological evolution. The genetic algorithm repeatedly modifies a population of individual solutions. At each step, the genetic algorithm selects individuals from the current population to be parents and uses them to produce the children for the next generation.

Over successive generations, the population "evolves" toward an optimal solution. You can apply the genetic algorithm to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic, or highly nonlinear.

Code:

import random

# Number of individuals in each generation

POPULATION\_SIZE = 100

# Valid genes

GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP

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

# Target string to be generated

TARGET = "UPG College Student "

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,

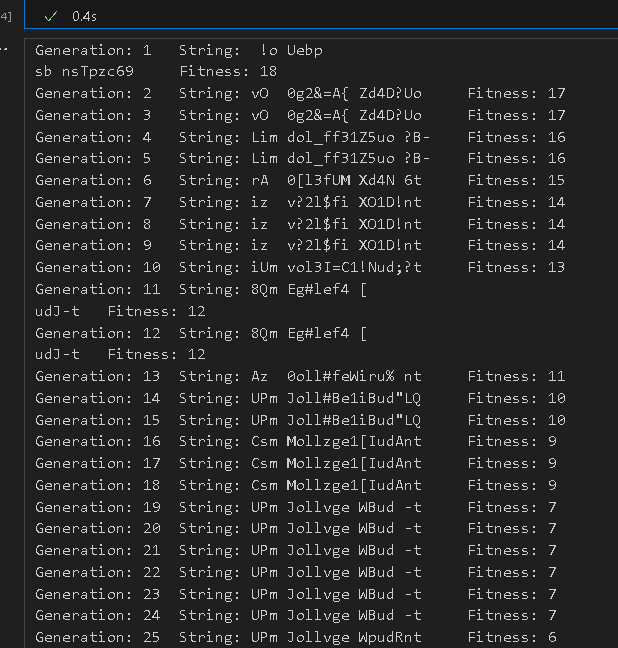
"".join(population[0].chromosome),

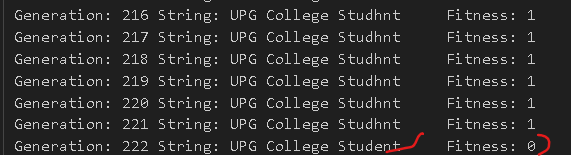
population[0].fitness))

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

main()

Output:





B] Create two classes: City and Fitness using Genetic algorithm

Code:

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

from tkinter import Tk, Canvas, Frame, BOTH, Text

import math

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) + ")"

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

def createRoute(cityList):

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

return route

def initialPopulation(popSize, cityList):

population = []

for i in range(0, popSize):

population.append(createRoute(cityList))

return population

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)

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

def matingPool(population, selectionResults):

matingpool = []

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

index = selectionResults[i]

matingpool.append(population[index])

return matingpool

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

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

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

def mutatePopulation(population, mutationRate):

mutatedPop = []

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

mutatedInd = mutate(population[ind], mutationRate)

mutatedPop.append(mutatedInd)

return mutatedPop

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

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

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()

def main():

cityList = []

for i in range(0,25):

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

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

main()

Output:

