

Practical 1a

Aim: Design a simple linear neural network model.

```
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((net>=0)&(net<=1)):  
    out =net  
else:  
    out=1  
print("net=",net)  
print("output=",out)
```

1b: Calculate the output of neural net using both binary and bipolar sigmoidal function.

For the network shown in the figure 1, calculate the net input to output neuron.

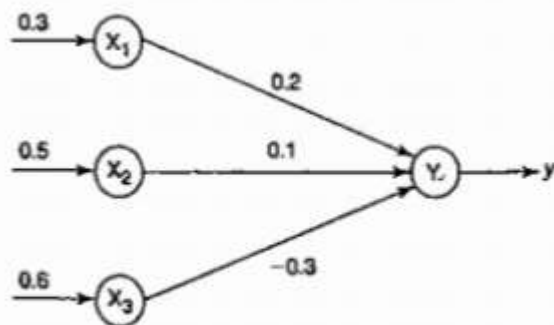


Figure 1 Neural net.

Solution : The given neural net consist of three input neurons and one output neuron.

The inputs and weight are

$$[x_1, x_2, x_3] = [0.3, 0.5, 0.6]$$

$$[w_1, w_2, w_3] = [0.2, 0.1, -0.3]$$

The net input can be calculated as

$$\begin{aligned} Y_{in} &= x_1w_1 + x_2w_2 + x_3w_3 \\ &= 0.3*0.2 + 0.5*0.1 + 0.6*(-0.3) \\ &= -0.07 \end{aligned}$$

Code :

```
# number of elements as input
n = int(input("Enter number of elements : "))

# In[2]:
print("Enter the inputs")
inputs = [] # creating an empty list for inputs

# iterating till the range
for i in range(0, n):
    ele = float(input())
    inputs.append(ele) # adding the element

print(inputs)

# In[3]:
print("Enter the weights")
# creating an empty list for weights
weights = []

# iterating till the range
for i in range(0, n):
    ele = float(input())
    weights.append(ele) # adding the element

print(weights)

# In[4]:
print("The net input can be calculated as  $Y_{in} = x_1w_1 + x_2w_2 + x_3w_3$ ")

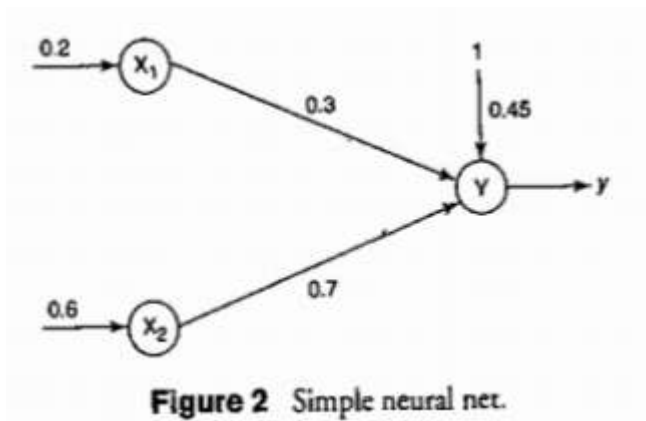
# In[5]:
Yin = []
for i in range(0, n):
    Yin.append(inputs[i]*weights[i])
print(round(sum(Yin),3))
```

Output :

```
Enter number of elements : 3
Enter the inputs
0.3
0.5
0.6
[0.3, 0.5, 0.6]
Enter the weights
0.2
0.1
-0.3
[0.2, 0.1, -0.3]
The net input can be calculated as  $Y_{in} = x_1w_1 + x_2w_2 + x_3w_3$ 
-0.07
```

1. Problem statement :

1. Calculate the net input for the network shown in Figure 2 with bias included in the network.



Solution: The given net consists of two input neurons, a bias and an output neuron. The inputs are

$[x_1, x_2] = [0.2, 0.6]$ and the weights are $[w_1, w_2] = [0.3, 0.7]$. Since the bias is included $b = 0.45$ and bias input x_0 is equal to 1, the net input is calculated as

$$\begin{aligned} Y_{in} &= b + x_1w_1 + x_2w_2 \\ &= 0.45 + 0.2 \times 0.3 + 0.6 \times 0.7 \\ &= 0.45 + 0.06 + 0.42 = 0.93 \end{aligned}$$

Therefore $y_m = 0.93$ is the net input.

Code :

```
n = int(input("Enter number of elements : "))

print("Enter the inputs:")
```

```
inputs = [] # creating an empty list for inputs
for i in range(0, n):
    ele = float(input())
    inputs.append(ele) # adding the element
print(inputs)

print("Enter the weights:")
weights = []
for i in range(0, n):
    ele = float(input())
    weights.append(ele) # adding the element
print(weights)

b=float(input("Enter bias value:"))
print("The net input can be calculated as  $Y_{in} = b + x_1w_1 + x_2w_2$ ")

Yin = []
for i in range(0, n):
    Yin.append(inputs[i]*weights[i])
print(round((sum(Yin)+b),3))

Enter number of elements : 2
Enter the inputs:
0.2
0.6
[0.2, 0.6]
Enter the weights:
0.3
0.7
[0.3, 0.7]
Enter bias value:0.45
The net input can be calculated as  $Y_{in} = b + x_1w_1 + x_2w_2$ :
0.93
```

Practical 2a:

Aim: Implement AND/NOT function using McCulloch-Pits neuron (use binary data representation).

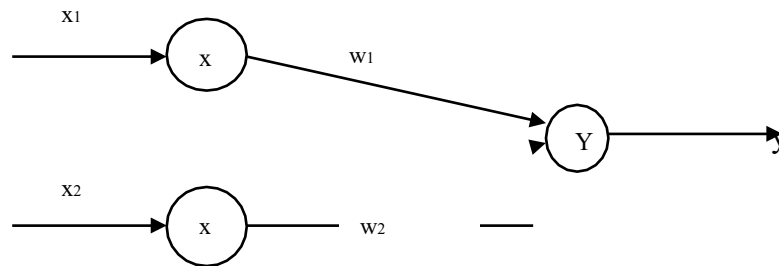
Solution:

In the case of AND/NOT function, the response is true if the first input is true and the second input is false. For all the other variations, the response is false. The truth table for ANDNOT function is given in Table below.

Truth Table:

x1	x2	y
0	0	0
0	1	0
1	0	1
1	1	0

The given function gives an output only when $x_1 = 1$ and $x_2 = 0$. The weights have to be decided only after the analysis. The net can be represent as shown in figure below:



Neural net (weights fixed after analysis).

Case 1: Assume that both weights w_1 and w_2 . are excitatory, i.e.,

$$w_1 = w_2 = 1$$

Then for the four inputs calculate the net input using

$$y_{ij} = x_1 w_1 + x_2 w_2$$

For inputs

$$(1, 1), y_{ij} = 1 \times 1 + 1 \times 1 = 2$$

$$(1, 0), y_{ij} = 1 \times 1 + 0 \times 1 = 1$$

$$(0, 1), y_{ij} = 0 \times 1 + 1 \times 1 = 1$$

$$(0, 0), y_{ij} = 0 \times 1 + 0 \times 1 = 0$$

From the calculated net inputs, it is not possible to fire the neuron from input (1, 0) only. Hence, J- weights are not suitable.

Assume one weight as excitatory and the other as inhibitory, i.e.,

$$w_1 = 1, w_2 = -1$$

Now calculate the net input. For the inputs

$$(1,1), y_{in} = 1 \times 1 + 1 \times -1 = 0$$

$$(1,0), y_{in} = 1 \times 1 + 0 \times -1 = 1$$

$$(0,1), y_{in} = 0 \times 1 + 1 \times -1 = -1$$

$$(0, 0), y_{in} = 0 \times 1 + 0 \times -1 = 0$$

From the calculated net inputs, now it is possible to fire the neuron for input (1, 0) only by fixing a threshold of 1, i.e., $\theta \geq 1$ for Y unit. Thus,

$$w_1 = 1, w_2 = -1; \theta \geq 1$$

Note: The value is calculated using the following:

$$\theta \geq nw - p$$

$$\theta \geq 2 \times 1 - 1$$

$$\theta \geq 1$$

Thus, the output of neuron Y can be written as

$$y = f(y_{in}) = \begin{cases} 0 & \text{if } y_{in} \geq 1 \\ 1 & \text{if } y_{in} < 1 \end{cases}$$

Code:

```
# enter the no of inputs
num_ip = int(input("Enter the number of inputs : "))

#Set the weights with value 1
w1 = 1
w2 = 1
print("For the ", num_ip , " inputs calculate the net input using  $yin = x1w1 + x2w2$  ")
x1 = []
x2 = []
for j in range(0, num_ip):
    ele1 = int(input("x1 = "))
    ele2 = int(input("x2 = "))
    x1.append(ele1)
    x2.append(ele2)
print("x1 = ",x1)
print("x2 = ",x2)

n = x1 * w1
m = x2 * w2

Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] + m[i])
print("Yin = ",Yin)
#Assume one weight as excitatory and the other as inhibitory, i.e.,
Yin = []
for i in range(0, num_ip):
    Yin.append(n[i] - m[i])
print("After assuming one weight as excitatory and the other as inhibitory Yin = ",Yin)

#From the calculated net inputs, now it is possible to fire the neuron for input (1, 0)
#only by fixing a threshold of 1, i.e.,  $\theta \geq 1$  for Y
unit. #Thus,  $w1 = 1$ ,  $w2 = -1$ ;  $\theta \geq 1$ 

Y=[]

for i in range(0, num_ip):
    if(Yin[i]>=1):
        ele= 1
        Y.append(ele)
    if(Yin[i]<1):
        ele= 0
```

```
Y.append(ele)
print("Y = ",Y)
Output:
```

```
Enter the number of inputs : 4
For the 4 inputs calculate the net input using  $y_{in} = x_1w_1 + x_2w_2$ 

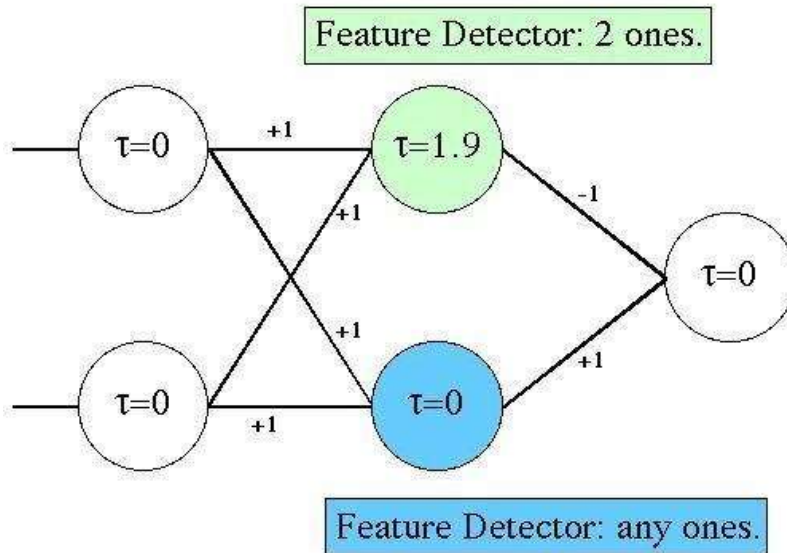
x1 = 0
x2 = 0
x1 = 0
x2 = 1
x1 = 1
x2 = 0
x1 = 1
x2 = 1
x1 = [0, 0, 1, 1]
x2 = [0, 1, 0, 1]
Yin = [0, 1, 1, 2]
After assuming one weight as excitatory and the other as inhibitory Yin = [0, 1, -1, 0]
Y = [0, 1, 0, 0]

In [14]: |
```


Practical 2b:

Aim: Generate XOR function using McCulloch-Pitts neural net

XOR Network



The XOR (exclusive or) function is defined by the following truth table:

Input1	Input2	XOR Output
0	0	0
0	1	1
1	0	1
1	1	0

#Getting weights and threshold value

```
import numpy as np
```

```
print('Enter weights')
```

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

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

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

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

SOFT COMPUTING TECHNIQUES

```
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)
```

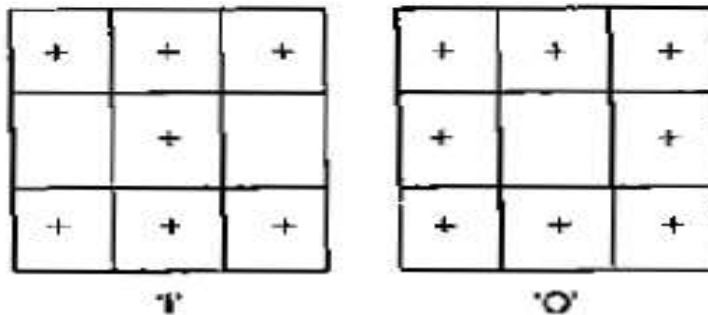
```
print(w22)
print("weights of Neuron Y")
print(v1)
print(v2)
print("Threshold value")
print(theta)
```

```
Enter weights
Weight w11=1
weight w12=-1
Weight w21=-1
weight w22=1
weight v1=1
weight v2=1
Enter Threshold Value
theta=1
z1 [ 0 -1  1  0]
z2 [ 0  1 -1  0]
yin [0. 1. 1. 0.]
Output of Net
y [0 1 1 0]
z [0 1 1 0]
McCulloch-Pitts Net for XOR function
Weights of Neuron Z1
1
-1
weights of Neuron Z2
-1
1
weights of Neuron Y
1
1
Threshold value
1
```

Practical NO. 3

3a) Aim: Write a program to implement Hebb's rule.

Using the Hebb rule, find the weights required to perform the following classifications of the given input patterns shown in Figure 16. The pattern is shown as 3×3 matrix form in the squares. The "+" symbols represent the value "1" and empty squares indicate "-1." Consider "I" belongs to the members of class (so has target value 1) and "O" does not belong to the members of class (so has target value -1).



```
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 bias 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)

bias=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("new wt =", wtnew)
print("Bias value",b)

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)
```

```
First input with target =1
new wt = [ 1  1  1 -1  1 -1  1  1  1]
Bias value 1
Second input with target =-1
new wt = [ 0  0  0 -2  2 -2  0  0  0]
Bias value 0
```

3b) Aim: Write a program to implement of delta

```
rule. #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)

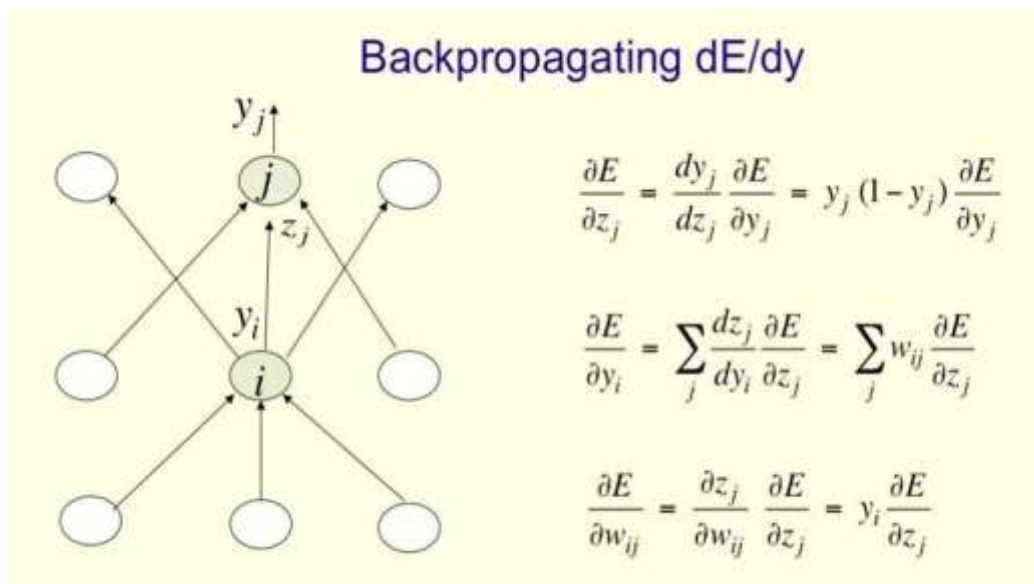
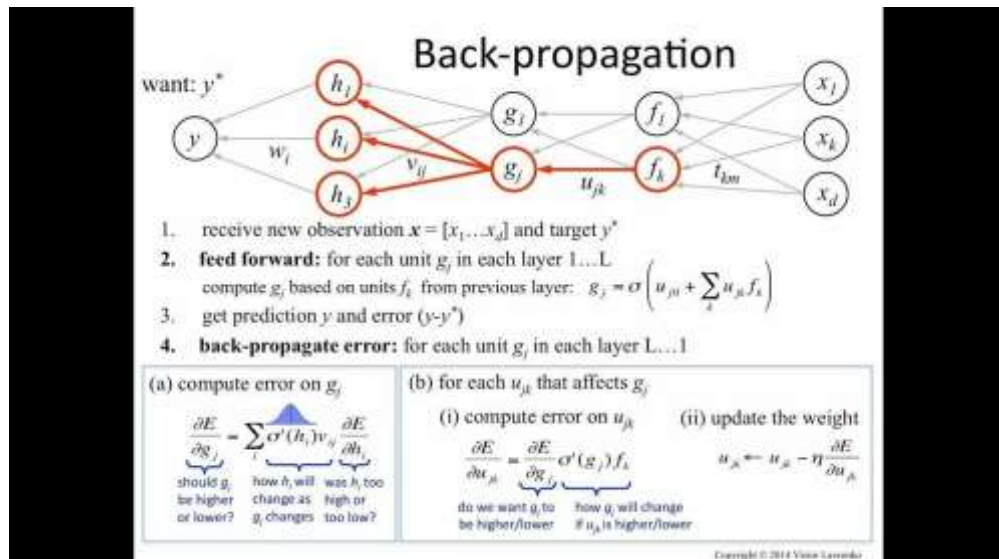
print("actual",actual)

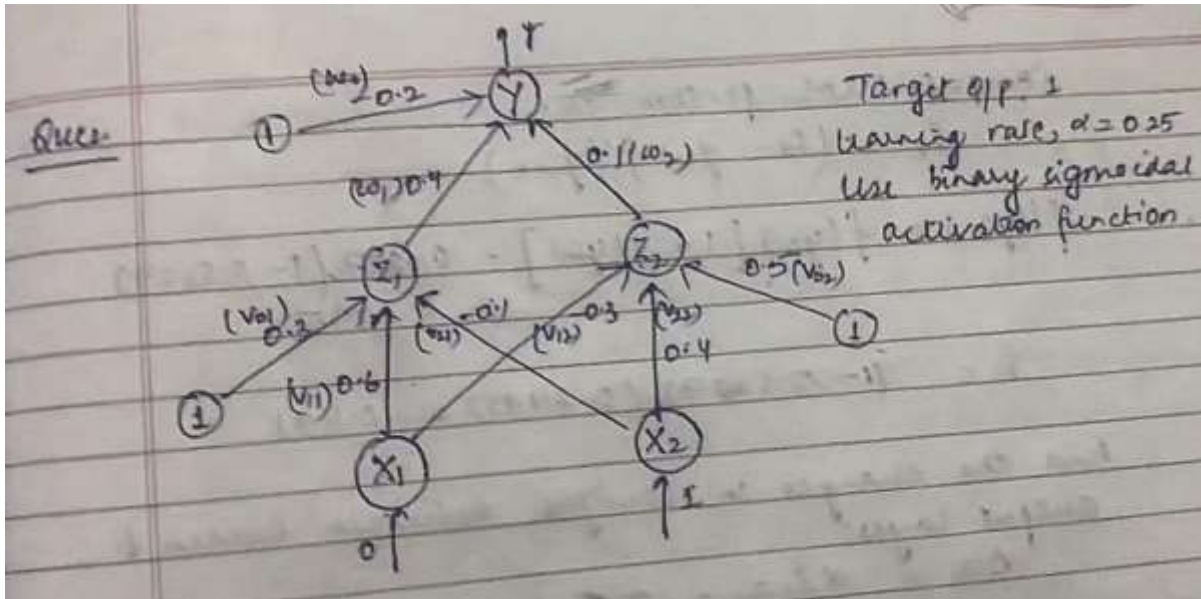
print("desired",desired)
```

```
Initial inputs:1
Initial inputs:1
Initial inputs:1
Initial weights:1
Initial weights:1
Initial weights:1
Desired output:2
Desired output:3
Desired output:4
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 3. 4.]
weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
*****
Final output
corrected weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
```


Practical 4a:

Aim: Write a program for Back Propagation Algorithm





```
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("v=",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)
```

```
z1= 0.2
```

```
calculate net input to z2 layer
```

```
z2= 0.9
```

```
Apply activation function to calculate output
```

```
z1= 0.5498
```

```
z2= 0.7109
```

```
calculate net input to output layer
```

```
yin= 0.09101
```

```
calculate net output
```

```
y= 0.5227368084248941
```

```
dk 0.11906907074145694
```

```
compute error portion in delta
```

```
din1= 0.04762762829658278
```

```
din2= 0.011906907074145694
```

```
error in delta
```

```
fzin1 0.24751996
```

```
fzin2 0.20552119000000002
```

```
d1= 0.011788788650865037
```

```
d2= 0.0024471217110978417
```

```
Changes in weights between input and hidden layer
```

```
dv11= 0.0
```

```
dv21= 0.0029471971627162592
```

```
dv01= 0.0029471971627162592
```

```
dv12= 0.0
```

```
dv22= 0.0006117804277744604
```

```
dv02= 0.0006117804277744604
```

```
Final weights of network
```

```
v= [0.6 0.3]
```

```
v2 [-0.1 0.4]
```

```
w= [-0.17 0.42 0.12]
```

```
bias b1= 0.30294719716271623 b2= 0.5006117804277744
```

Practical 4b

Aim: Write a Program For Error Back Propagation Algorithm (Ebpa) Learning

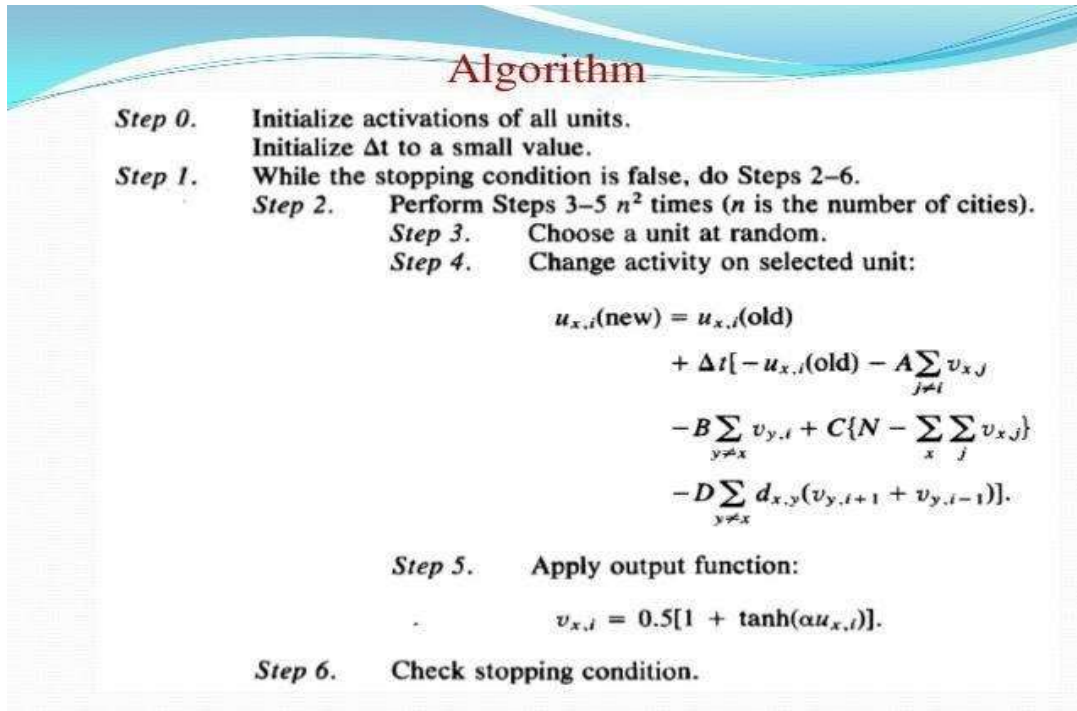
```
import math
a0=-1
t=-1
w10=float(input("Enter weight first network"))
b10=float(input("Enter base first network:"))
w20=float(input("Enter weight second network:"))
b20=float(input("Enter base second network:"))
c=float(input("Enter learning coefficient:"))
n1=float(w10*c+b10)
a1=math.tanh(n1)
n2=float(w20*a1+b20)
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)
```

Enter weight first network:12
Enter base first network:35
Enter weight second network:23
Enter base second network:45
Enter learning coefficient:11
The updated weight of first n/w w11= 12.0
The uploaded weight of second n/w w21= 23.0
The updated base of first n/w b10= 35.0
The updated base of second n/w b20= 45.0

Practical 5a:

Aim: Write a program for Hopfield Network.



The image shows a slide titled "Algorithm" with a blue and white wavy header. The slide contains a list of steps for the Hopfield Network algorithm, including initialization, a while loop for stopping conditions, and specific steps for updating unit activities and applying an output function. Mathematical formulas are provided for the update rule and the output function.

Algorithm

Step 0. Initialize activations of all units.
Initialize Δt to a small value.

Step 1. While the stopping condition is false, do Steps 2–6.

Step 2. Perform Steps 3–5 n^2 times (n is the number of cities).

Step 3. Choose a unit at random.

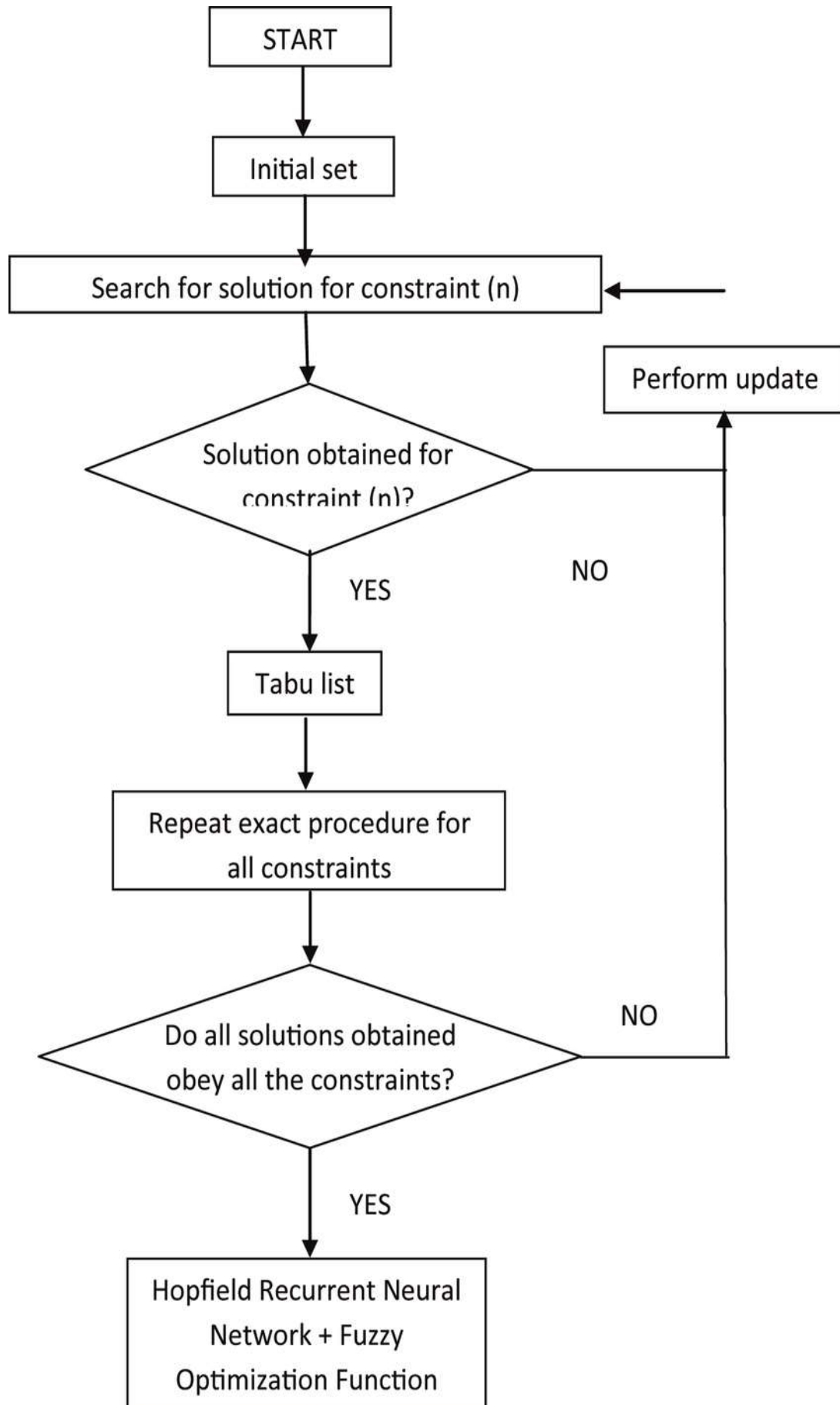
Step 4. Change activity on selected unit:

$$u_{x,i}(\text{new}) = u_{x,i}(\text{old}) + \Delta t \left[-u_{x,i}(\text{old}) - A \sum_{j \neq i} v_{x,j} - B \sum_{y \neq x} v_{y,i} + C \left\{ N - \sum_x \sum_j v_{x,j} \right\} - D \sum_{y \neq x} d_{x,y} (v_{y,i+1} + v_{y,i-1}) \right]$$

Step 5. Apply output function:

$$v_{x,i} = 0.5[1 + \tanh(\alpha u_{x,i})].$$

Step 6. Check stopping condition.




```
#include "hop.h"
neuron::neuron(int *j)
{
    inti;
    for(i=0;i<4;i++)
    {
        weightv[i]= *(j+i);
    }
}
int neuron::act(int m, int *x)
{
    inti;
    int a=0;
    for(i=0;i<m;i++)
    {
        a += x[i]*weightv[i];
    }
    return a;
}
int network::threshld(int k)
{
    if(k>=0)
        return (1);
    else
        return (0);
}
network::network(int a[4],int b[4],int c[4],int d[4])
{
    nrn[0] = neuron(a) ;
    nrn[1] = neuron(b) ;
    nrn[2] = neuron(c) ;
    nrn[3] = neuron(d) ;
}

void network::activation(int *patrn)
{
    inti,j;
    for(i=0;i<4;i++)
    {
        for(j=0;j<4;j++)
        {
            cout<<"\n nrn["<<i<<"].weightv["<<j<<"] is "
                <<nrn[i].weightv[j];
        }
        nrn[i].activation = nrn[i].act(4,patrn);
        cout<<"\nactivation is "<<nrn[i].activation;
        output[i]=threshld(nrn[i].activation);
    }
}
```

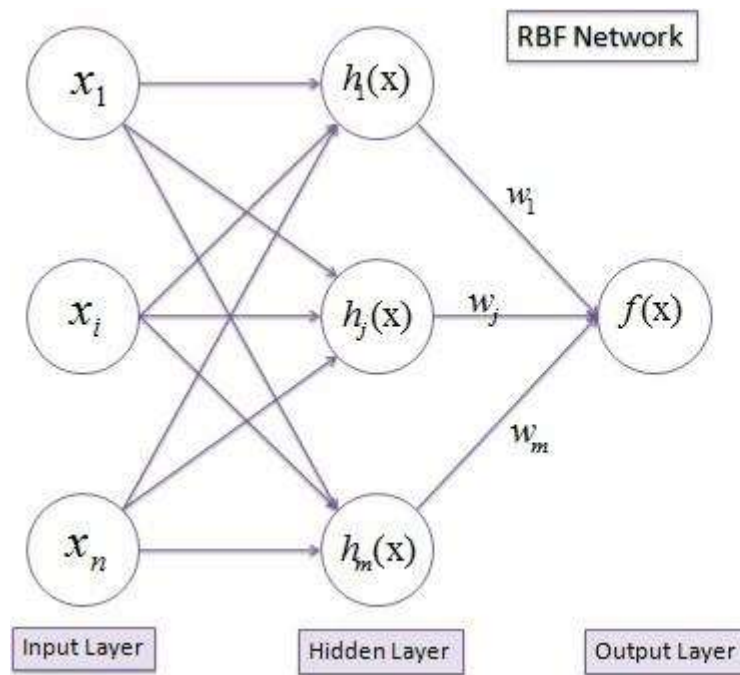
```
        cout<<"\noutput value is "<<output[i]<<"\n";
    }
}
void main ()
{
    int patrn1[]= {1,0,1,0},i;
    int wt1[]= {0,-3,3,-3};
    int wt2[]= {-3,0,-3,3};
    int wt3[]= {3,-3,0,-3};
    int wt4[]= {-3,3,-3,0};
    cout<<"\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER
    OF";
    cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD
    RECALL THE";
    cout<<"\nPATTERNS 1010 AND 0101 CORRECTLY.\n";
    //create the network by calling its constructor.
    // the constructor calls neuron constructor as many times as thenumber of
    // neurons in the network.
    network h1(wt1,wt2,wt3,wt4);
    //present a pattern to the network and get the activations of the neurons
    h1.activation(patrn1);
    //check if the pattern given is correctly recalled and give message
    for(i=0;i<4;i++)
    {
        if (h1.output[i] == patrn1[i])
            cout<<"\n pattern= "<<patrn1[i]<<
            " output = "<<h1.output[i]<<" component matches";
        else
            cout<<"\n pattern= "<<patrn1[i]<<
            " output = "<<h1.output[i]<<
            " discrepancy occurred";
    }
    cout<<"\n\n";
    int patrn2[]= {0,1,0,1};
    h1.activation(patrn2);
    for(i=0;i<4;i++)
    {
        if (h1.output[i] == patrn2[i])
            cout<<"\n pattern= "<<patrn2[i]<<
            " output = "<<h1.output[i]<<" component matches";
        else
            cout<<"\n pattern= "<<patrn2[i]<<
            " output = "<<h1.output[i]<<
            " discrepancy occurred";
    }
}
===== End code of main program=====
//Hop.h
//Single layer Hopfield Network with 4 neurons
```

```
#include <stdio.h>
#include <iostream.h>
#include <math.h>
class neuron
{
protected:
    int activation;
    friend class network;
public:
    intweightv[4];
    neuron() {};
    neuron(int *j) ;
    int act(int, int*);
};
class network
{
public:
    neuron nrn[4];
    int output[4];
    intthreshld(int) ;
    void activation(int j[4]);
    network(int*,int*,int*,int*);
};
```

Practical 5b:

Aim: Write a program for Radial Basis function

Radial Basis Function Networks (RBF)



$$f(x) = \sum_{j=1}^m w_j h_j(x)$$

$$h(x) = \exp\left(-\frac{(x-c)^2}{r^2}\right)$$

```
from scipy import *

from scipy.linalg import norm, pinv
from matplotlib import pyplot as plt

class RBF:

    def __init__(self, indim, numCenters, outdim):
        self.indim =indim
        self.outdim =outdim
        self.numCenters =numCenters
        self.centers =[random.uniform(-1, 1, indim) for i in range(numCenters)]
        self.beta =8
        self.W =random.random((self.numCenters, self.outdim))

    def _basisfunc(self, c, d):
        assert len(d) ==self.indim
        return exp(-self.beta *norm(c-d)**2)

    def _calcAct(self, X):
        # calculate activations of RBFs
        G =zeros((X.shape[0], self.numCenters), float)
        for ci, c in enumerate(self.centers):
            for xi, x in enumerate(X):
                G[xi,ci] =self._basisfunc(c, x)
        return G

    def train(self, X, Y):
        """ X: matrix of dimensions n x indim
            y: column vector of dimension n x 1 """
```

```
# choose random center vectors from training set
rnd_idx = random.permutation(X.shape[0])[:self.numCenters]
self.centers = [X[i,:] for i in rnd_idx]

print("center", self.centers)

# calculate activations of RBFs
G = self._calcAct(X)
print (G)

# calculate output weights (pseudoinverse)
self.W = dot(pinv(G), Y)

def test(self, X):
    """ X: matrix of dimensions n x indim """

    G = self._calcAct(X)
    Y = dot(G, self.W)
    return Y

if __name__ == '__main__':
    # ___ - 1D Example _____ -
    n = 100

    x = mgrid[-1:1:complex(0,n)].reshape(n, 1)
    # set y and add random noise
    y = sin(3*(x+0.5)**3-1)
    # y += random.normal(0, 0.1, y.shape)

    # rbf regression
```

```
rbf=RBF(1, 10, 1)
rbf.train(x, y)
z =rbf.test(x)

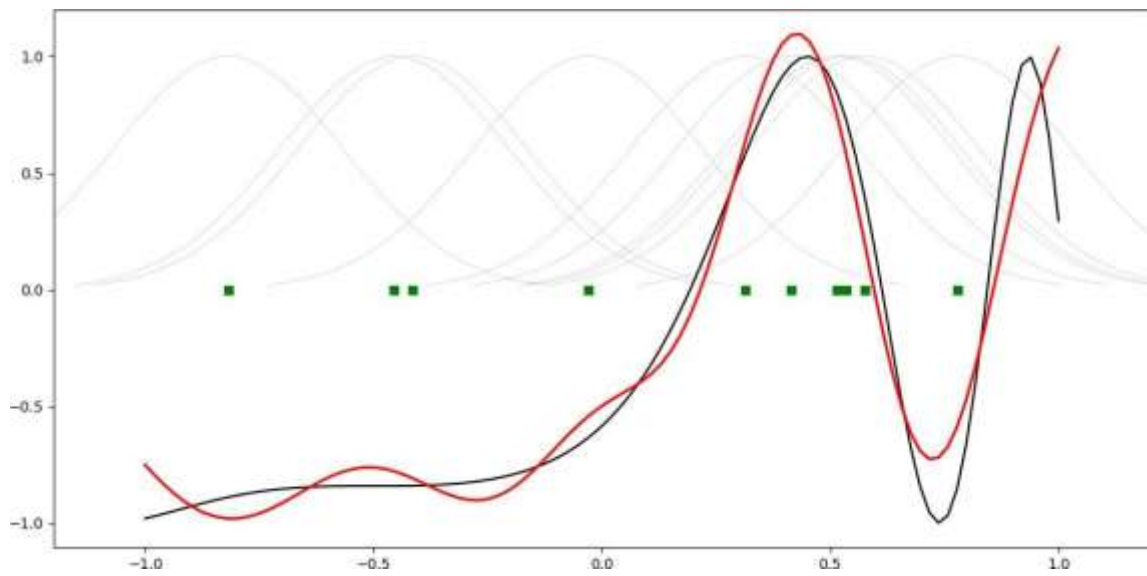
# plot original data
plt.figure(figsize=(12, 8))
plt.plot(x, y, 'k-')

# plot learned model
plt.plot(x, z, 'r-', linewidth=2)

# plot rbfs
plt.plot(rbf.centers, zeros(rbf.numCenters), 'gs')

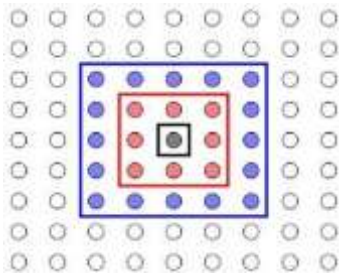
for c in rbf.centers:
    # RF prediction lines
    cx =arange(c-0.7, c+0.7, 0.01)
    cy =[rbf._basisfunc(array([cx_]), array([c])) for cx_ in cx]
    plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

plt.xlim(-1.2, 1.2)
plt.show()
```

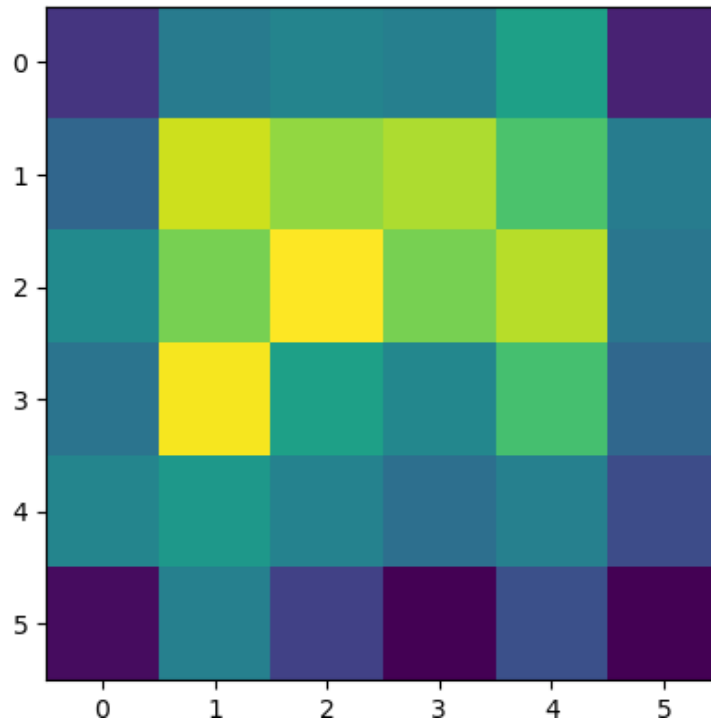


Practical 6a:

Aim:Self-Organizing Maps



```
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())
plt.show()
```



Practical 7a:

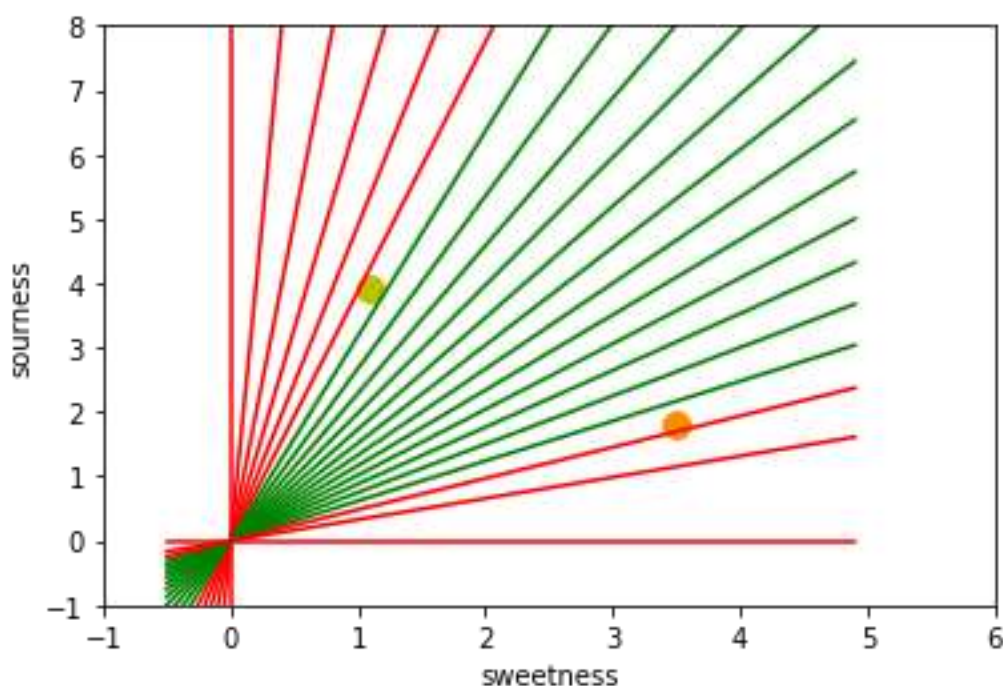
Aim: Line Separation

```
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()
```



Practical 7b:

Aim: Hopfield Network model of associative memory

The Hopfield model (226), consists of a network of N neurons, labeled by a lower index i , with $1 \leq i \leq N$. Similar to some earlier models (335; 304; 549), neurons in the Hopfield model have only two states. A neuron i is 'ON' if its state variable takes the value $S_i = +1$ and 'OFF' (silent) if $S_i = -1$. The dynamics evolves in discrete time with time steps Δt . There is no refractoriness and the duration of a time step is typically not specified. If we take $\Delta t = 1\text{ms}$, we can interpret $S_i(t) = +1$ as an action potential of neuron i at time t . If we take $\Delta t = 500\text{ms}$, $S_i(t) = +1$ should rather be interpreted as an episode of high firing rate.

Neurons interact with each other with weights w_{ij} . The input potential of neuron i , influenced by the activity of other neurons is

$$h_i(t) = \sum_j w_{ij} S_j(t). \quad (17.2)$$

The input potential at time t influences the probabilistic update of the state variable S_i in the next time step:

$$\text{Prob}\{S_i(t+\Delta t) = +1 | h_i(t)\} = g(h_i(t)) = g\left(\sum_j w_{ij} S_j(t)\right) \quad (17.3)$$

where g is a monotonically increasing gain function with values between zero and one. A common choice is $g(h) = 0.5[1 + \tanh(\beta h)]$ with a parameter β . For $\beta \rightarrow \infty$, we have $g(h) = 1$ for $h > 0$ and zero otherwise. The dynamics are therefore deterministic and summarized by the update rule

$$S_i(t+\Delta t) = \text{sgn}[h_i(t)] \quad (17.4)$$

For finite β the dynamics are stochastic. In the following we assume that in each time step all neurons are updated synchronously (parallel dynamics), but an update scheme where only one neuron is updated per time step is also possible.

Source code:

```
%matplotlib inline
from neurodynex.hopfield_network import network, pattern_tools, plot_tools

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]

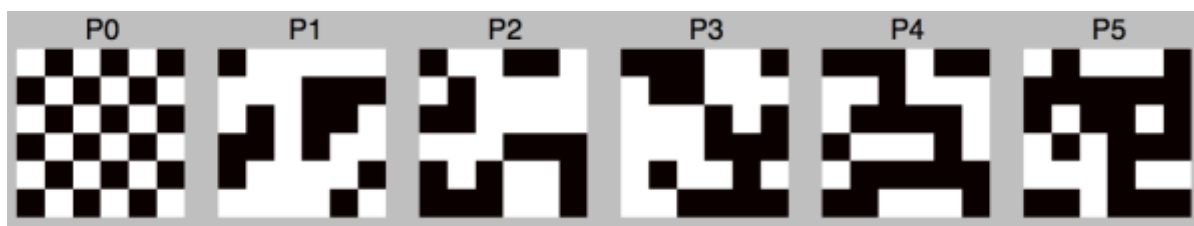
# add random patterns to the list
pattern_list.extend(factory.create_random_pattern_list(nr_patterns=3, on_probability=0.5))
plot_tools.plot_pattern_list(pattern_list)
# how similar are the random patterns and the checkerboard? Check the overlaps
overlap_matrix = pattern_tools.compute_overlap_matrix(pattern_list)
plot_tools.plot_overlap_matrix(overlap_matrix)

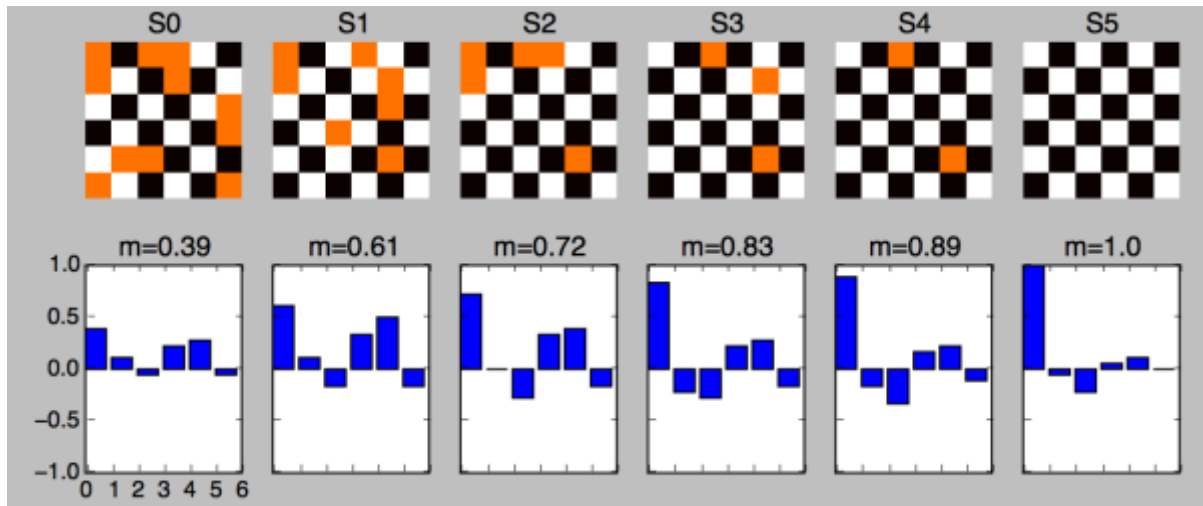
# let the hopfield network "learn" the patterns. Note: they are not stored
# explicitly but only network weights are updated !
hopfield_net.store_patterns(pattern_list)

# 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")
```





Practical 8a:

Aim: Membership and Identity operators in, not in.

Python program to illustrate
Finding common member in list
without using 'in' operator

```
# Define a function() that takes two lists
def overlapping(list1,list2):
    c=0
    d=0
    for i in list1:
        c+=1
    for i in list2:
        d+=1
    for i in range(0,c):
        for j in range(0,d):
            if(list1[i]==list2[j]):
                return 1
    return 0
list1=[1,2,3,4,5]
list2=[6,7,8,9]
if(overlapping(list1,list2)):
    print("overlapping")
else:
    print("not overlapping")
```

Python program to illustrate
Finding common member in list
without using 'in' operator

```
# Define a function() that takes two lists
def overlapping(list1,list2):
```

```
c=0
d=0
for i in list1:
    c+=1
for i in list2:
    d+=1
for i in range(0,c):
    for j in range(0,d):
        if(list1[i]==list2[j]):
            return 1
    return 0
list1=[1,2,3,4,5]
list2=[6,7,8,9]
if(overlapping(list1,list2)):
    print("overlapping")
else:
    print("not overlapping")
.
```

Practical 8b: Membership and Identity Operators is, is not

```
# Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
    print ("true")
else:
    print ("false")
.
```

```
# Python program to illustrate the
# use of 'is not' identity operator
x = 5.2
if (type(x) is not int):
    print ("true")
else:
    print ("false")
```

Practical 9a:

Find the ratios using fuzzy logic

```
pip install fuzzywuzzy
```

```
# 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 fuzzysforfuzzys"
```

```
s2 = "I am loving fuzzysforfuzzys"
```

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

```
print ("FuzzyWuzzyPartialRatio: ", fuzz.partial_ratio(s1, s2))
```

```
print ("FuzzyWuzzyTokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))
```

```
print ("FuzzyWuzzyTokenSetRatio: ", fuzz.token_set_ratio(s1, s2))
```

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

```
# for process library,
```

```
query = 'fuzzys for fuzzys'
```

```
choices = ['fuzzy for fuzzy', 'fuzzy fuzzy', 'g. for fuzzys']
```

```
print ("List of ratios: ")
```

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

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

```
FuzzyWuzzy Ratio: 86
```

```
FuzzyWuzzyPartialRatio: 86
```

```
FuzzyWuzzyTokenSortRatio: 86
```

```
FuzzyWuzzyTokenSetRatio: 87
```

```
FuzzyWuzzyWRatio: 86
```

```
List of ratios:
```

```
[('g. for fuzzys', 95), ('fuzzy for fuzzy', 94), ('fuzzy fuzzy', 86)]
```

```
Best among the above list: ('g. for fuzzys', 95)
```


Practical 9b:

Aim: Solve Tipping Problem using fuzzy logic

Fuzzy Control Systems: The Tipping Problem

Creating the Tipping Controller Using the skfuzzy control API

We can use the `skfuzzy` control system API to model this. First, let's define fuzzy variables

Code:

```
import numpy as np
import skfuzzy as fuzz
from skfuzzy import control as ctrl

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')

quality.automf(3)
service.automf(3)

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])

quality['average'].view()

service.view()
tip.view()

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()

tipping_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])

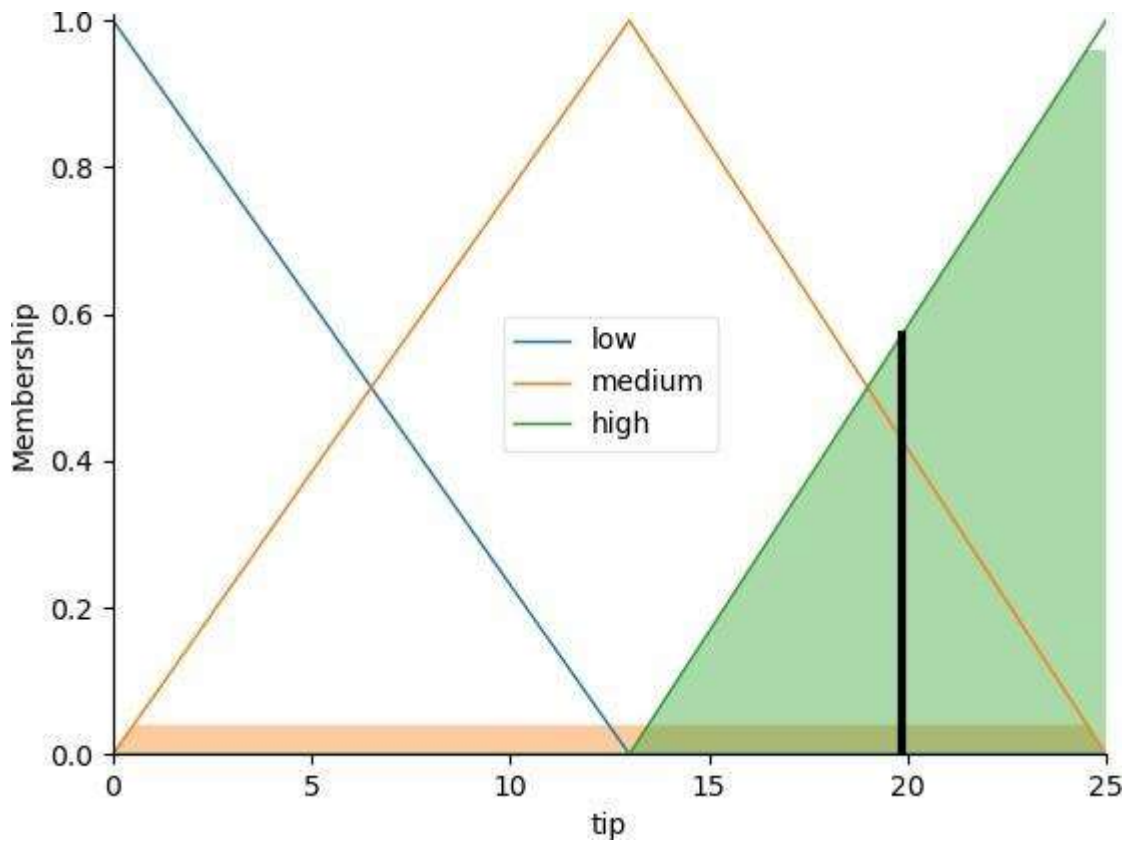
tipping = ctrl.ControlSystemSimulation(tipping_ctrl)

# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API
tipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

# Crunch the numbers
```

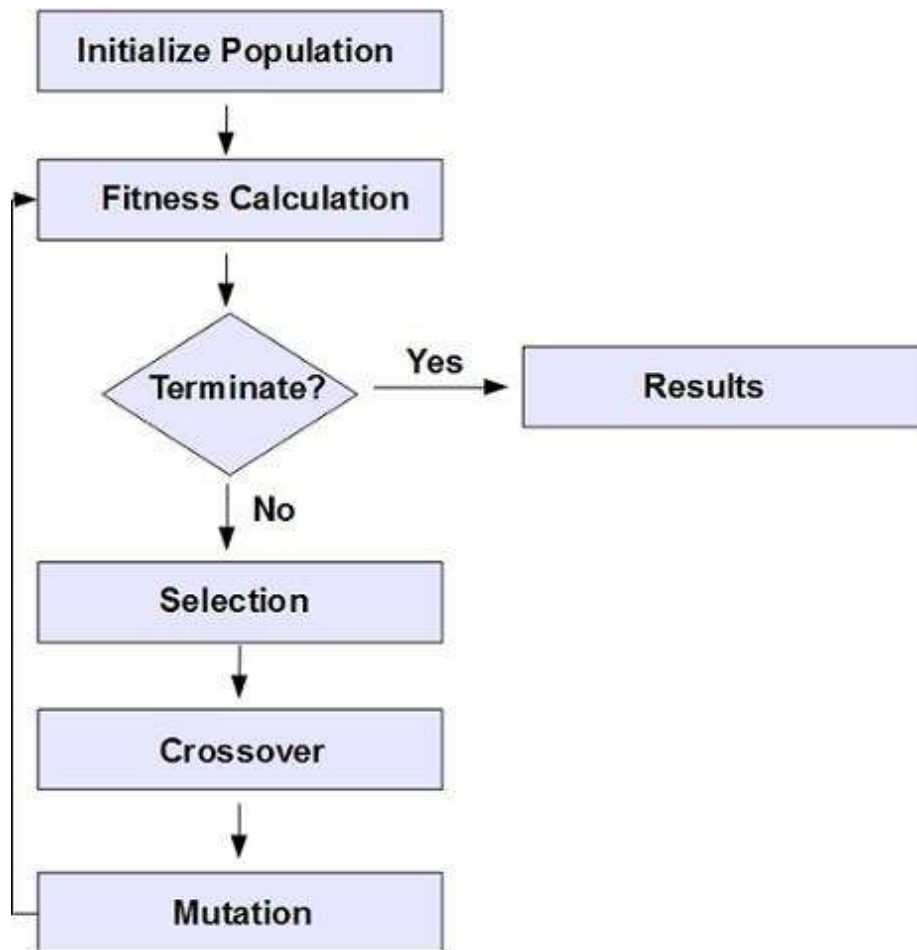
```
tipping.compute()
```

```
print tipping.output['tip']  
tip.view(sim=tipping)
```

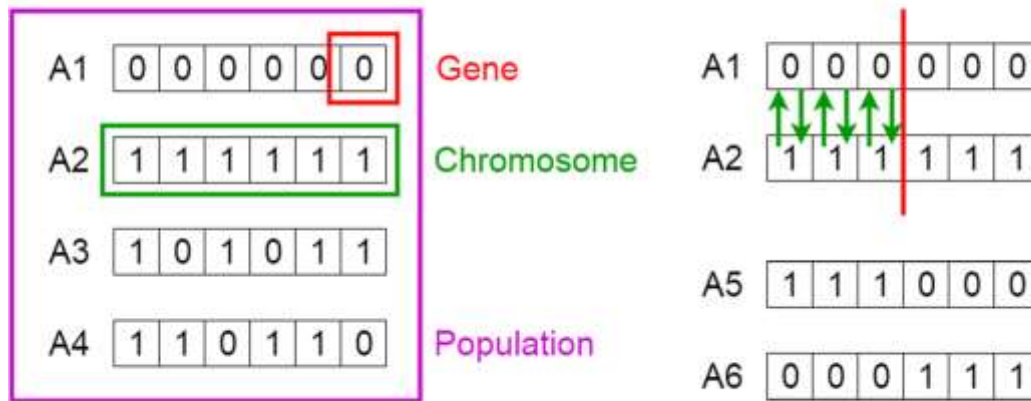


The resulting suggested tip is ** 19.8476**

Practical 10:
Aim: Implementation of simple genetic algorithm



Genetic Algorithms



```
import random
```

```
# Number of individuals in each generation  
POPULATION_SIZE =100
```

```
# Valid genes  
GENES = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ  
QRSTUVWXYZ 1234567890, .-:;!\"#%&/()=?@${[]}\""
```

```
# Target string to be generated  
TARGET = "Mithilesh Chauhan"
```

```
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 fitness 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:

```

Generation: 1  String: tO{"-?=jH[k8=B4]Oe@}    Fitness:
Generation: 2  String: tO{"-?=jH[k8=B4]Oe@}    18
Generation: 3  String: .#lRWf9k_Ifslw #O$k_      Fitness:
Generation: 4  String: .-lRq?9mHqk3Wo]3rek_      18
Generation: 5  String: .-lRa?9mHak3Wo]3rek_      Fitness: 17
Generation: 6  String: A#ldW) #llkslwcVek)    Fitness: 14
Generation: 7  String: A#ldW) #llkslwcVek)    Fitness: 14
Generation: 8  String: (, o x _x%Rs=, 6Peek3    Fitness: 13
.
.
.
Generation: 29 String: I lope Geeks#o,          Fitness: 3
Generation: 30 Geeks String: I                  Fitness:
Generation: 31 loMeGeeksfoBGeeks String: I      2
Generation: 32 love Geeksfo0Geeks String: I      Fitness: 1
Generation: 33 love Geeksfo0Geeks String: I      Fitness: 1
Generation: 34 love Geeksfo0Geeks String: I      Fitness: 1

```

Practical 10b:

Aim: Create two classes: City and Fitness using Genetic algorithm

First create a City class that will allow us to create and handle our cities.

Create Population

<https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35>

```
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)  
  
if __name__ == '__main__':  
    main()
```

