**Name: Adarsh Kushwaha**

**Roll no:**

**Subject: Soft Computing(Practicals)**

**Practical 1a**

**Aim: Design a simple linear neural network model.**

**Code:**

x = float(input("enter the vlaue of x :"))

w = float(input("enter the value of w :"))

b = float(input("enter the vlaue of bias :"))

net = (x\*w+b)

if(net<0):

out = 0

elif((net>=0) & (net <=1)):

out = net

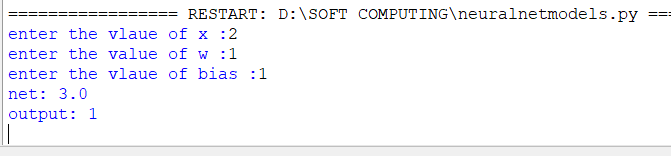
else:

out = 1

print("net:",net)

print("output:",out)

**Output:**

****

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

**Code:**

**n = int(input("enter the number of inputs:"))**

**inputs=[]**

**print("enter the inputs")**

**for i in range(0,n):**

**elements = float(input())**

**inputs.append(elements)**

**print(inputs)**

**print("enter the weights :")**

**weights =[]**

**for i in range(0,n):**

**weight = float(input())**

**weights.append(weight)**

**print(weights)**

**print("The net input can be calculated as Yin = x1w1+x2w2+x3w3")**

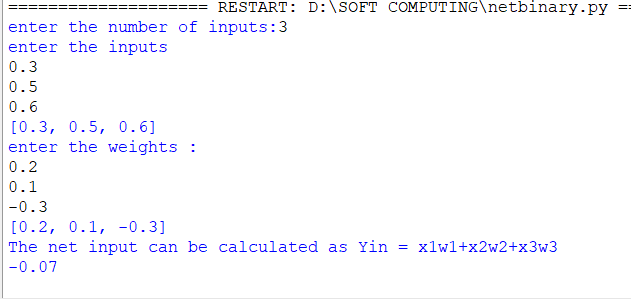
**Yin = []**

**for i in range(0,n):**

**Yin.append((inputs[i]\*weights[i]))**

**print(round(sum(Yin),3))**

**Output:**

****

**Code:**

**n = int(input("enter the number of inputs:"))**

**inputs=[]**

**print("enter the inputs")**

**for i in range(0,n):**

**elements = float(input())**

**inputs.append(elements)**

**print(inputs)**

**print("enter the weights :")**

**weights =[]**

**for i in range(0,n):**

**weight = float(input())**

**weights.append(weight)**

**print(weights)**

**#b = float(input("enter the bias value :"))**

**print("The net input can be calculated as Yin = x1w1+x2w2+x3w3")**

**Yin = []**

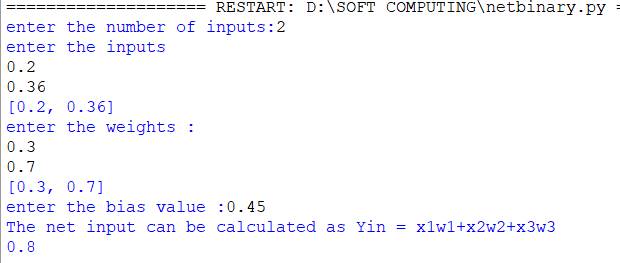
**for i in range(0,n):**

**Yin.append((inputs[i]\*weights[i]))**

**print(round(sum(Yin),3))**

**#print(round((sum(Yin)+b),1))**

**Output:**

****

**Practical 2a:**

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

**Code:**

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

**w1=2**

**w2=1**

**print("For the ",num\_ip,"inputs calculat 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)**

**Yin =[]**

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

**Yin.append(m[i]-n[i])**

**print("After assuming one weight as excitatory and the other as inhibitory Yin= ",Yin)**

**Y=[]**

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

**if(Yin[i]>=1):**

**ele=1**

**Y.append(ele)**

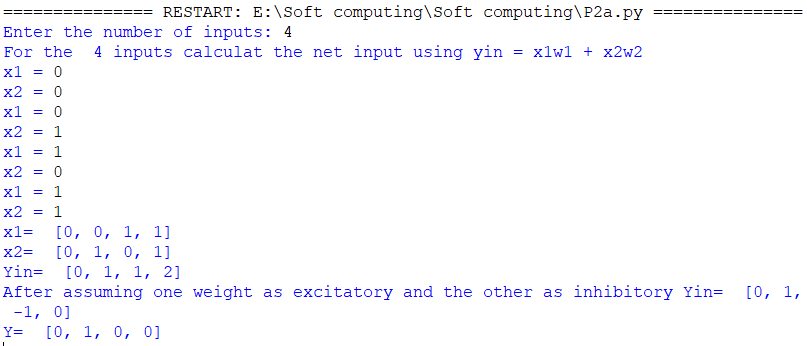
**if(Yin[i]<1):**

**ele=0**

**Y.append(ele)**

**print("Y= ",Y)**

**Output:**

****

**Practical 2b**

**Aim: Generate XOR function using McCulloch-Pitts neural net.**

**Code:**

**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 ='))**

**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 the another set of weights and Threshold values")**

**w11= input("weights w11 = ")**

**w12= input("weights w12 = ")**

**w21= input("weights w21 = ")**

**w22= input("weights w22 = ")**

**v1= input("weights v1= ")**

**v2= input("weights 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")**

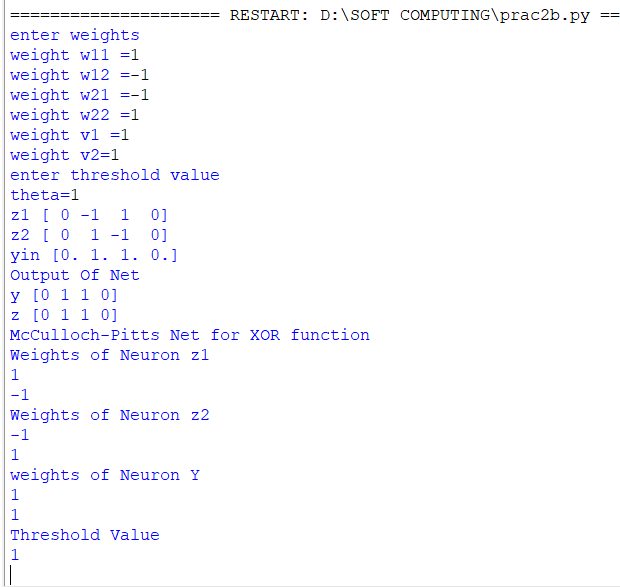
**print(v1)**

**print(v2)**

**print("Threshold Value")**

**print(theta)**

**Output:**



**Practical 3a**

**Aim: Write a program to implement Hebb’s rule.**

**Code:**

**import numpy as np**

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

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

**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("new wt =",wtnew)**

**print("bias vlaue",b)**

**print("second input with target = -1")**

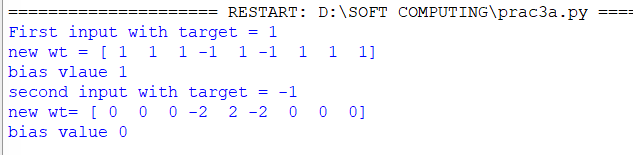
**for i in range(0,9):**

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

**b= b+y[1]**

**print("new wt=",wtnew)**

**print('bias value',b) Output:**



**Practical 3b**

**Aim: Write a program to implement of delta rule.**

**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("Desired Output:"))**

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

**actual = x\*weights**

**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)**

**print("actual",actual)**

**print("desired",desired)**

**print("\*"\*30)**

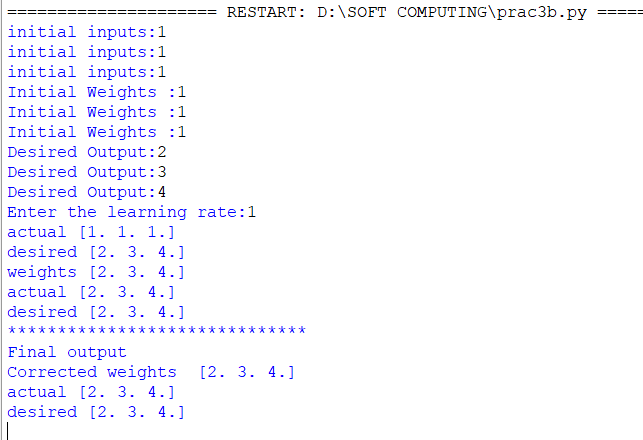
**print("Final output")**

**print("Corrected weights ", weights)**

**print("actual",actual)**

**print("desired",desired)**

**output:**



**Practical 4a**

**Aim: Write a program for Back Propagation Algorithm**

**Code:**

**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**

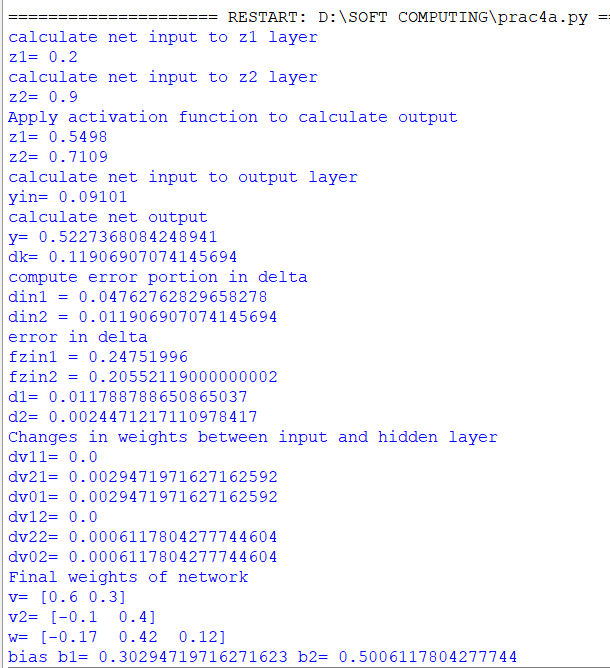
**b2=b2+dv02**

**w[0]=w[0]+dw0**

**print("w=",w)**

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

**output:**

****

**Practical 4b**

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

**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\*c+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)**

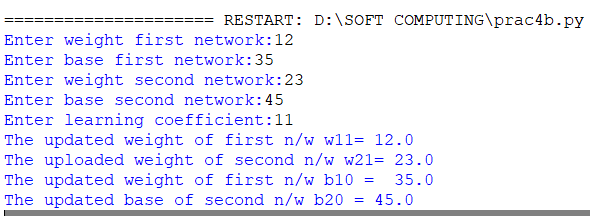
**print("The updated weight of first n/w w11=",w11)**

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

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

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

**Output:**



**Practical 6a**

**Aim:Self-Organizing Maps**

**Code:**

**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]]**

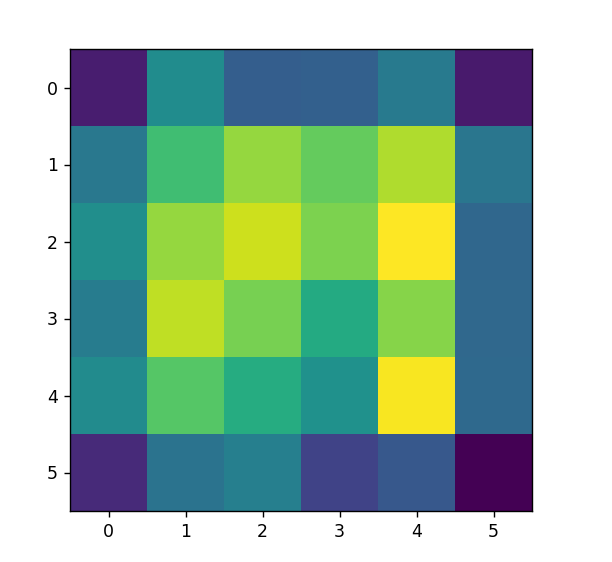
**som =MiniSom(6,6,4,sigma=0.3,learning\_rate=0.5)**

**som.train\_random(data,100)**

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

**plt.show()**

**Output:**

****

**Practical 7a**

**Aim: Line 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",""]**

**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)**

**Y=slope\*X**

**results=[]**

**for point in points:**

**results.append(dist4line1(\*point))**

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

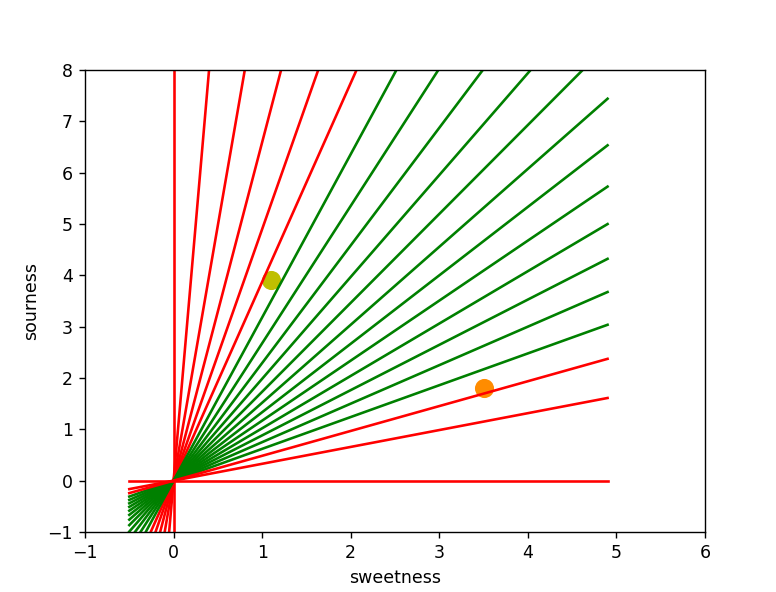
**ax.plot(X,Y,"g-")**

**else:**

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

**plt.show()**

**Output:**

****

**Practical 7b**

**Aim: Hopfield Network model of associative memory**

**Code:**

**import matplotlib.pyplot as plt**

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

**# Parameters**

**pattern\_size = 10 # Define the size of the patterns**

**nr\_neurons = pattern\_size \*\* 2**

**# Create an instance of the Hopfield network**

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

**# Instantiate a pattern factory**

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

**# Create a checkerboard pattern and a random pattern list**

**checkerboard = factory.create\_checkerboard()**

**pattern\_list = [checkerboard]**

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

**# Plot the patterns**

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

**# Compute and plot the overlap matrix**

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

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

**# Store patterns in the Hopfield network**

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

**# Create a noisy version of the checkerboard pattern**

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

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

**# Run the Hopfield network with monitoring**

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

**# Reshape the states into patterns**

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

**# Plot the state sequence and overlap**

**plot\_tools.plot\_state\_sequence\_and\_overlap(**

**states\_as\_patterns,**

**pattern\_list,**

**reference\_idx=0,**

**suptitle="Network Dynamics"**

**)**

**# Show plots**

**plt.show()**

**Output:**

**ERROR**

**Practical 8a**

**Aim: Membership and Identity operators in, not in.**

**Code:**

**#Aim: Membership and Identity operators in, not in.**

**# Python program to illustrate**

**# Finding common member in list**

**# without using 'in' operator**

**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")**

**Output:**

**not overlapping**

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

**Code:**

**# 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")**

**Output:**

**True**

**True**

**Practical 9a**

**Aim: Find the ratios using fuzzy logic**

**Code:**

**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("FuzzyWuzzyPartial Ratio:",fuzz.partial\_ratio(s1,s2))**

**print("FuzzyWuzzyTokenSort Ratio:",fuzz.token\_sort\_ratio(s1,s2))**

**print("FuzzyWuzzyTokenSet Ratio:",fuzz.token\_set\_ratio(s1,s2))**

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

**#for process library**

**query="fuzzy for fuzzys"**

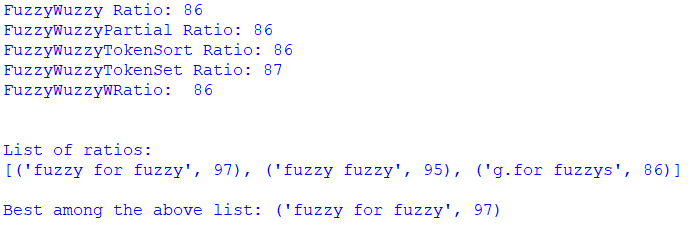
**choices=['fuzzy for fuzzy','fuzzy fuzzy','g.for fuzzys']**

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

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

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

**Output:**

****

**Practical 9b: Solve Tipping Problem using fuzzy logic**

**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(qulity['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)**

**tipping.input['quality']=6.5**

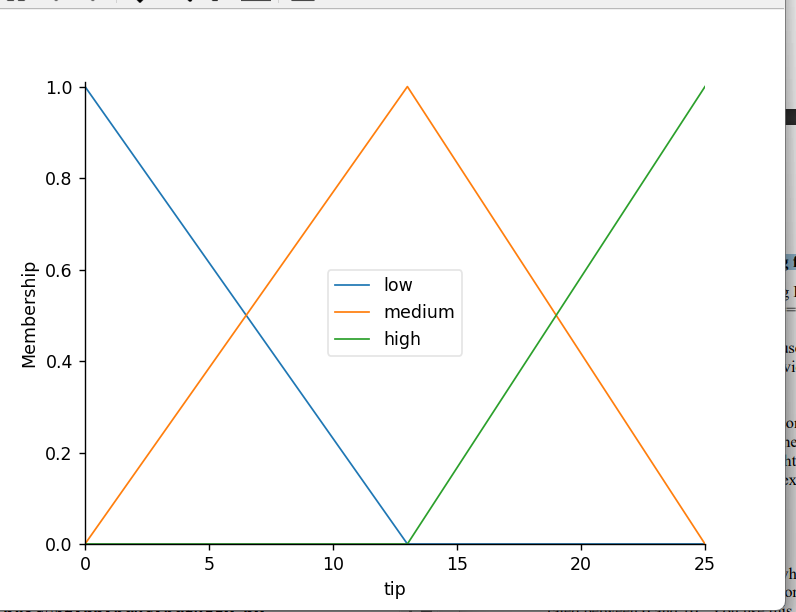
**tipping.input['service']=9.8**

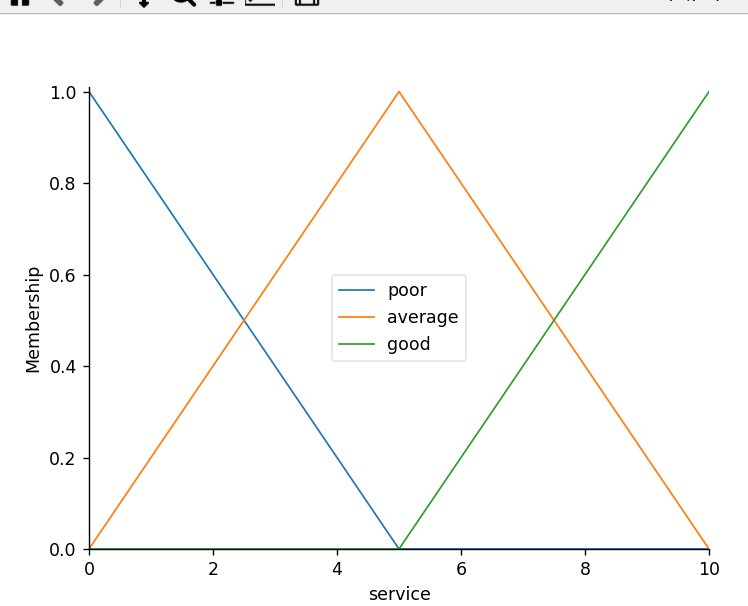
**tipping.compute()**

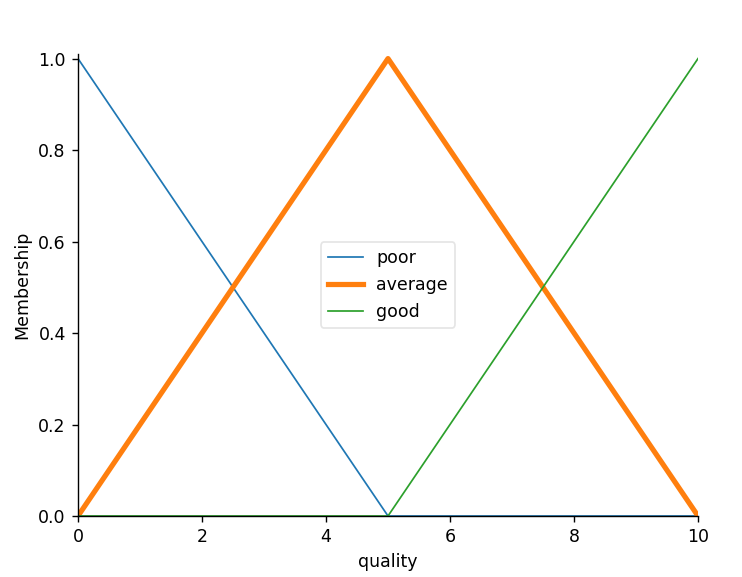
**print(tipping.output['tip'])**

**tip.views(sim=tipping)**

**Output:**

****

****

****