M.Sc.IT Information

Technology

Semester- I

Soft Computing

Submitted By

Prashant Dnyaneshwar Shingade

SEAT NO: ______

YEAR 2021-2022

Submitted in Partial Fulfillment of requirement for

qualifyingM.Sc.IT Part I (Sem-I) Examination

UNIVERSITY OF
MUMBAI

VIDYA VIKAS EDUCATION SOCIETY'S

VIKAS COLLEGE OF ARTS, SCIENCE &

COMMERCEVIKHROLI (E)-400 083

Vidya Vikas Education Society's

VIKAS COLLEGE OF ARTS,SCIENCE & COMMERCE

Affliated to

University of MumbaiRE-ACCREDITED 'A'

GRADE BY NAAC (WITH CGPA 3.15

ISO 9001 : 2008 CERTIFIED

Vikas High School Marg, Kannamwar Nagar No 2, Vikhroli (E), Mumbai – 400083

Dr. R. K. Patra Hon' ble: **Shri P. M. Raut**

Principal Chairman. V. V. Edu. Society

Email : vikascollegeprincipal@gmail.com
www.vikascollege.org

This is to certify that **Prashant Dnyaneshwar Shingade** Student of M.Sc.IT Part I(Sem-I) with Seat No._and college enrolled Roll no.**211532** has satisfactorily completed the practical work in

Information Technology Laboratory for the Course in the program of INFORMATION TECHNOLOGY from the UNIVERSITY OF MUMBAI for
the academic year 2021-2022 .

Subject In-Charge:__________________ HOD: __________________

Examiner: __________________

Practical No	Details	Date	Sign
1	Implement the following:		
a	Design a simple linear neural network model.		
b	Calculate the output of neural net using both binary and bipolar sigmoidal function.		
2	Implement the following:		
a	Generate AND/NOT function using McCulloch-Pitts neural net.		
b	Generate XOR function using McCulloch-Pitts neural net.		
3	Implement the Following		
a	Write a program to implement Hebb's rule.		
b	Write a program to implement of delta rule.		
4	Implement the Following		
a	Write a program for Back Propagation Algorithm		
b	Write a program for error Backpropagation algorithm.		
5.	Implement the Following		
a	Write a program for Hopfield Network.		
b	Write a program for Radial Basis function		
6.	Implement the Following		
a	Kohonen Self organizing map		

b	Adaptive resonance theory		
7.	Implement the Following		
a	Write a program for Linear separation.		
b	Write a program for Hopfield network model for associative memory		
8.	Implement the Following		
a	Membership and Identity Operators | in, not in,		
b.	Membership and Identity Operators is, is not		
9.	Implement the Following		
a	Find ratios using fuzzy logic		
b	Solve Tipping problem using fuzzy logic		
10.	Implement the Following		
a	Implementation of Simple genetic algorithm		
b	Create two classes: City and Fitness using Genetic algorithm		

P

ractical 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(n
```

```
et<
0):
    out=0 elif((net>=0)&(net<=1)):
ut =netelse:
    out=1 print("net=",net)
print("output=",out)
```

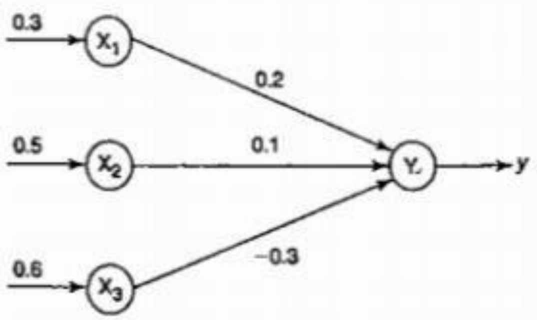


Figure 1 Neural net.

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.

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

$[x1, x2, x3] = [0.3, 0.5, 0.6]$

[w1, w2, w3] = [0.2, 0.1, -0.3]

The net input can be calculated as

$$
\begin{aligned}
Yin &= x1w1+x2w2+x3w3 \\
&= 0.3*0.2+0.5*0.1+0.6*(-0.3) \\
&= -0.07
\end{aligned}
$$

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
rangefor i
in
range(0,
n):
        ele = float(input())
        inputs.append(ele) # adding the

elementprint(inputs)

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

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

the elementprint(weights)
```

```
# In[4]:
print("The net input can be calculated as  Yin = x1w1 + x2w2 + x3w3")

# 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   Yin = x1w1 + x2w2 + x3w3
-0.07
```

1. **Problem statement :**

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

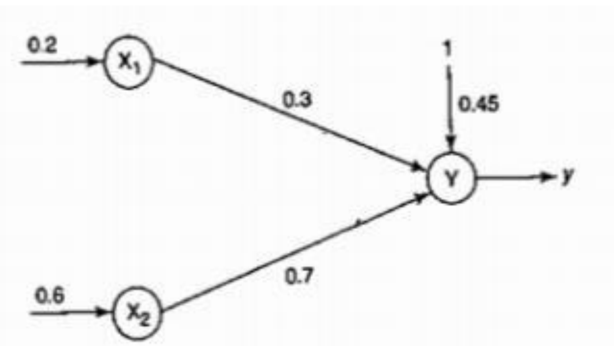


Figure 2 Simple neural net.

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 includedb = 0.45 and bias input x_o is equal to 1, the net input is calculated as

$$Yin = b + x_1W_1 + X_2W_2$$
$$= 0.45 + 0.2 \text{ X } 0.3 + 0.6 \text{ X } 0.7$$
$$= 0.45 + 0.06 + 0.42 = 0.93$$

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 inputsfor i in range(0,
n):
        ele = float(input())
        inputs.append(ele) # adding the
elementprint(inputs)

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

b=float(input("Enter bias value:"))
print("The net input can be calculated as  Yin = b + x1w1 + x2w2:")

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   Yin = b + x1w1 + x2w2:
0.93
```

Practical 2a:

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

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:

x_1	x_2	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 bedecided only after the analysis. The net can be represent as shown in figure below:

x_1

x w1

Y y

x_2

x w2

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

(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 form 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 byfixing 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

$y = f(y_{in}) = \{$

Code:

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

#Set the weights with value 1w1 = 1
w2 = 1

- if $y_{in} \geq 1$
- if $y_{in} < 1$

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("x
 1 = "))ele2
 =
 int(input("x
 2 = "))
 x1.append(
 ele1)

```
    x2.append(
    ele2)
print("x1 = ",x1)
print("x2 = ",x2)

n

=

x
1

*

w
1

m

=

x
2

*

w
2

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., θ ≥ 1 for Y unit.
#Thus, w1 = 1, w2 = -1; θ ≥ 1

Y=[]

for i in
    range(0,
    num_ip):
    if(Yin[i]
    >=1):
        e
        l
        e
        =

        1

        Y
        .
        a
        p
        p
        e
        n
        d
        (
        e
        l
        e
        )
    if(Yin[i]<1):
        e
        l
        e
        =

        0
```

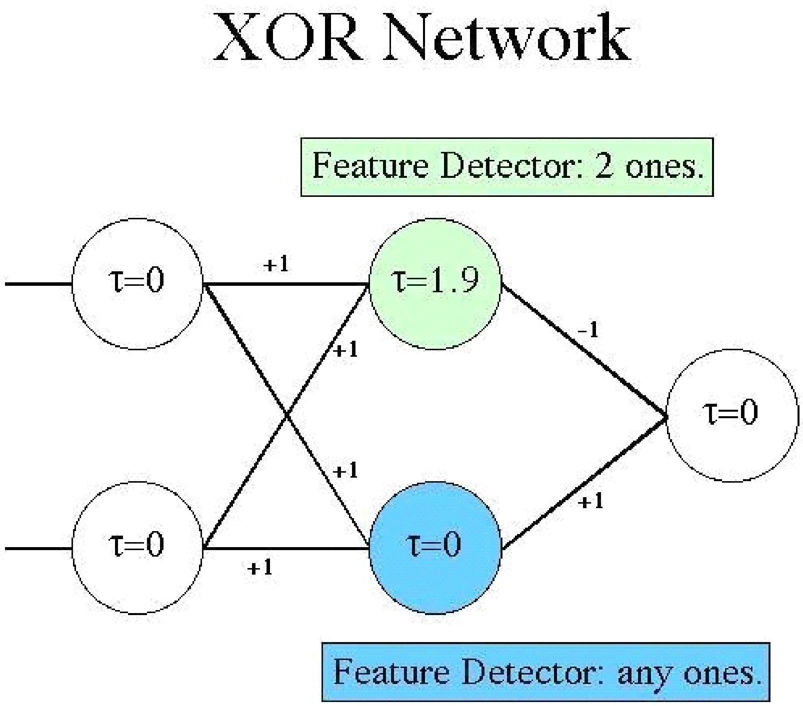
```
Y
.
a
p
p
e
n
d
(
e
l
e
)
print("Y = ",Y)
```

Output:

```
Enter the number of inputs : 4
For the  4  inputs calculate the net input using yin = x1w1 + x2w2

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



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 valueimport
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.arra
y([0, 1,
1, 0])
con=1
y1=np.ze
ros((4,))
y2=np.zeros((4,))
y=
np
.ze
ro
s((
4,)
)
w
```

```
hil
e
co
n=
=1
:
    zin1=np.zeros((4,))
    zin2=np.z
    eros((4,))
    zin1=x1*
    w11+x2*
    w21
    zin2=x1*
    w21+x2*
    w22

    print("z1",zin1)
    print
    ("z2
    ",zin
    2)
    for i
    in
    rang
    e(0,4
    ):
        if
            z
```

```
    i
    n
    1
    [
    i
    ]
    >
    =
    t
    h
    e
    t
    a
    :
    y
    1
    [
    i
    ]
    =
    1
else:
    y1[i]=0

if
    z
    i
    n
```

```
    2
    [
    i
    ]
    >
    =
    t
    h
    e
    t
    a
    :
    y
    2
    [
    i
    ]
    =
    1
  else:
    y2[i]=0
yin=
np.ar
ray([
])
yin=
y1*v
1+y2
```

*v2
for i
in
rang
e(0,4
):
    if
        y
        i
        n
        [
        i
        ]
        >
        =
        t
        h
        e
        t
        a
        :

        y
        [
        i
        ]
        =
        1

```
    else:
        y[i]=0

print("y
in",yin)
print('O
utput of
Net')
y=y.ast
ype(int)
print("y
",y)
print("z",z)

if
    np.arr
    ay_eq
    ual(y,
    z):
    con=0
else:
    print("Net is not learning enter another set of weights and
    Threshold value")w11=input("Weight w11=")
    w12=input("weigh
    t w12=")
    w21=input("Weigh
    t w21=")
    w22=input("weigh
```

```
t w22=")
v1=input("weight
v1=")
v2=input("weight
v2=")
theta=input("theta
=")

print("McCulloch-Pitts Net for
XOR function")print("Weights of
Neuron Z1")
p
r
i
n
t
(
w
1
1
)

p
r
i
n
t
(
```

w

2

1

)

print("weights of

Neuron Z2")

print(w12)

print(w22)

print("weights

of Neuron Y")

print(v1)

print(v2)

print("Thre

shold

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
```

P

ractical 3a. Aim: Write a program to

implement Hebb's rule.

The Hebbian Learning Rule is a learning rule that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. The rule builds on Hebbs's 1949 learning rule which states that the connections between two neurons might be strengthened if the neurons fire simultaneously. The Hebbian Rule works well as long as all the input patterns are orthogonal or uncorrelated. The requirement of orthogonality places serious limitations on the Hebbian Learning Rule. A more powerful learning rule is the delta rule, which utilizes the discrepancy between the desired and actual output of each output unit to change the weights feeding into it.

Hebb's rule with an analogy. Psychology and

neuroscience The Hebb's principle or Hebb's

rule

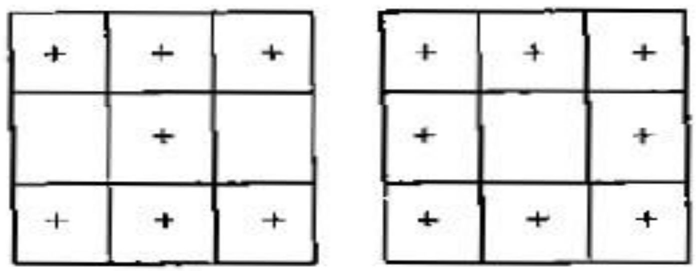
Hebb says that "when the axon of a cell A is close enough to excite a B cell and takes part on its activation in a repetitive and persistent way, some type of growth process or metabolic change takes place in one or both cells, so that increases the efficiency of cell A in the activationof B ".

'neurons that fire together wire together'

It is customary to be summarized as "neurons that fire together wire together".That is, the simultaneous activation of nearby neurons leads to an increase in the strength of synaptic connection between them.

It is important to note that the neurons must be previously connected, sufficiently close to one another, so that the synapse can be reinforced. Hebb's principle can be described as a method of determining how to alter the weights between model neurons. The weight between two neurons increases if the two neurons activate simultaneously, and reduces if they activate separately. Nodes that tend to be either both positive or both negative at the same time have strong positive weights, while those that tend to be opposite have strong negative weights.

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).



'I' 'O'

```
import
nump
y as
np
#first
patter
n
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])
#initial
ize
bais
value
b=0
#def
ine
targ
et
y=n
p.arr
ay([
```

```
1,-
1])

wtold=n
p.zeros(
(9,))
wtnew=
np.zeros
((9,))

wtnew=wtnew.astype(in
t)
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
value",b)

print("Second input
with target =-1")for i in
range(0,9):

    wtnew[i]=wtold[i]+x
2[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
```

P

ractical 3b:Aim: Write a program to

implement of delta rule. #supervised

learning

import

nump

y as

np

import

time

np.set_printoptions(prec

ision=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("In

    itial inputs:"))

```
for i in range(0,3):
    weights[i]=float(input("Initi
    al weights:"))

for i in range(0,3):
    desired[i]=float(input("Desi
    red output:"))

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

actual=x*weights
print("actual",actual)
print("desired",desired)

while True:
    if
        np.array_equal(d
        esired,actual):
        break    #no
        change
    else:

        for i in range(0,3):
            weights[i]=weights[i]+a*(desire
            d[i]-actual[i])

    actual=x*weights
    print("weights",weigh
    ts)
```

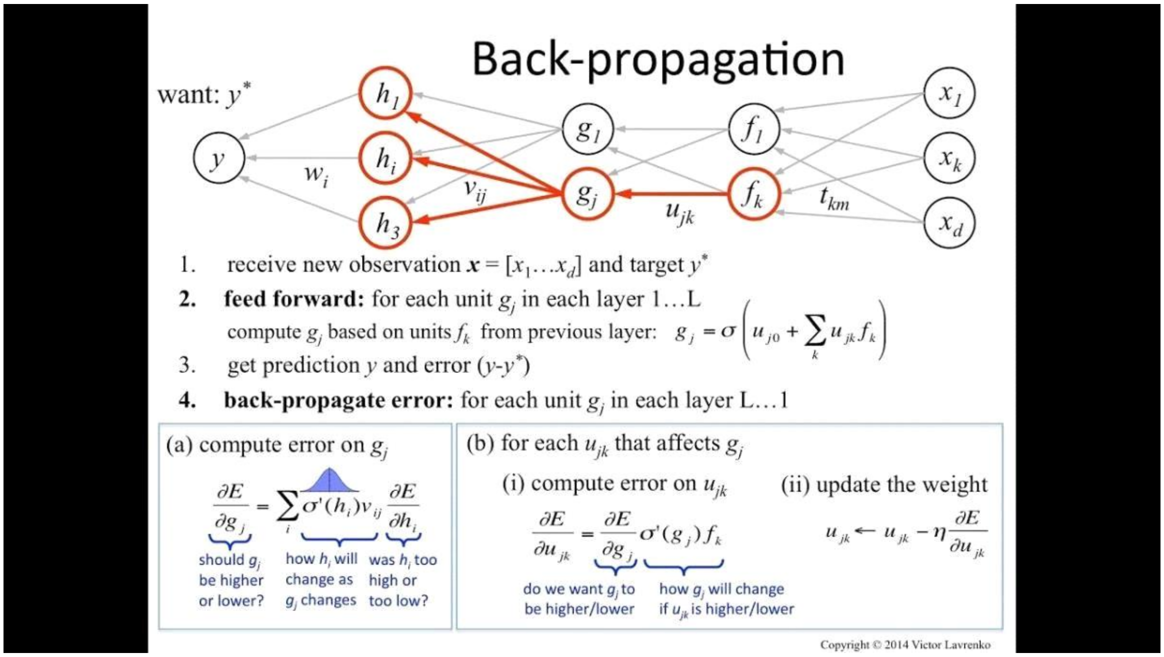
```
    print("actual",actual)
    print("desired",desire
    d)
print("*
"*30)
print("F
inal
output")
print("Corrected
weights",weights)
print("actual",actual)
print("desired",desire
d)
```

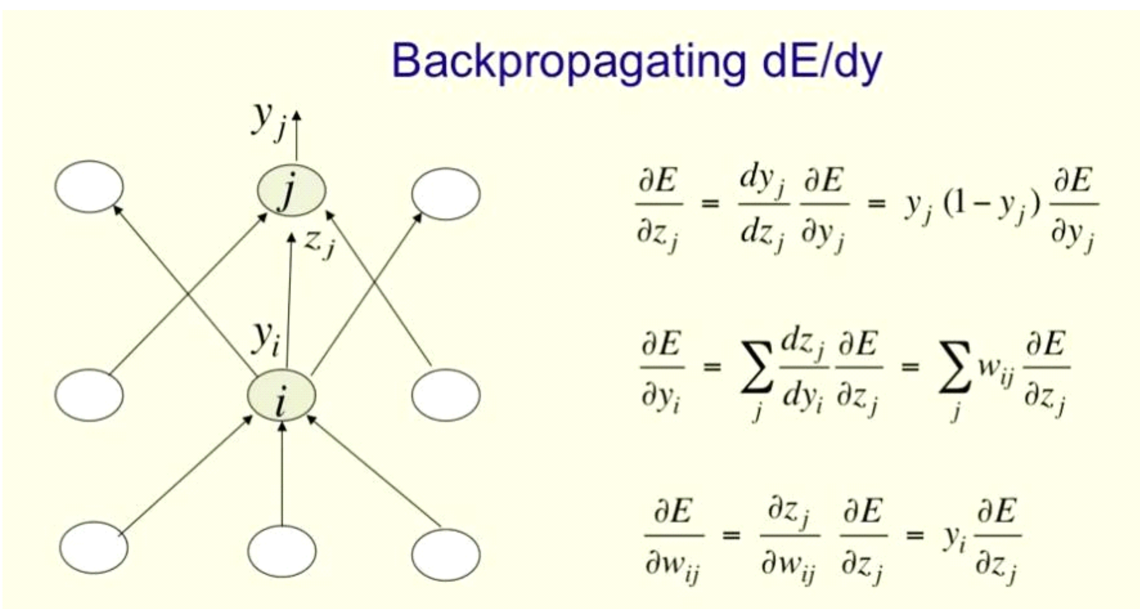
```
Initial inputs:1
Initial inputs:1
Initial inputs:1
Initial weights:1
Initial weights:1
Initial weights:1
Desired output:2
Desired output:3
Desired output:4
Enter learning rate:1
actual [1. 1. 1.]
desired [2. 3. 4.]
weights [2. 3. 4.]
actual [2. 3. 4.]
desired [2. 3. 4.]
******************************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

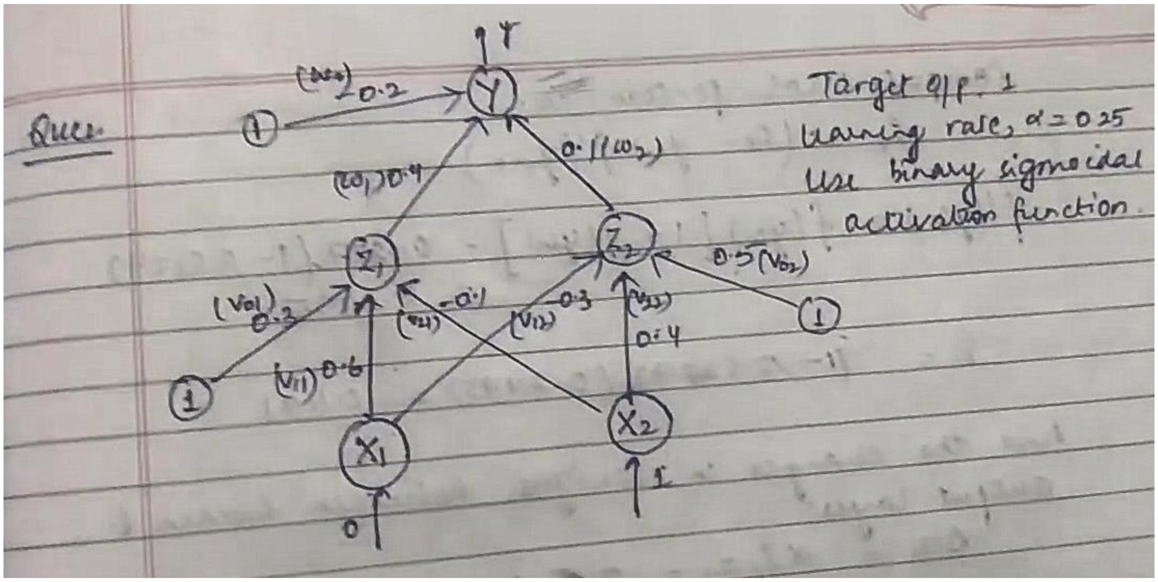
Pract

ical 4a: Aim: Write a program for Back

Propagation Algorithm


Back-propagation
want: y^*
y w_i h_1 h_i h_3 v_{ij} g_1 g_j u_{jk} f_1 f_k t_{km} x_1 x_k x_d
1. receive new observation $\boldsymbol{x} = [x_1 \ldots x_d]$ and target y^*
2. **feed forward:** for each unit g_j in each layer 1…L
compute g_j based on units f_k from previous layer: $g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right)$
3. get prediction y and error $(y\text{-}y^*)$
4. **back-propagate error:** for each unit g_j in each layer L…1
(a) compute error on g_j
$\underbrace{\frac{\partial E}{\partial g_j}}_{\text{should } g_j \text{ be higher or lower?}} = \sum_i \underbrace{\sigma'(h_i) v_{ij}}_{\text{how } h_i \text{ will change as } g_j \text{ changes}} \underbrace{\frac{\partial E}{\partial h_i}}_{\text{was } h_i \text{ too high or too low?}}$
(b) for each u_{jk} that affects g_j
(i) compute error on u_{jk}
$\frac{\partial E}{\partial u_{jk}} = \underbrace{\frac{\partial E}{\partial g_j}}_{\text{do we want } g_j \text{ to be higher/lower}} \underbrace{\sigma'(g_j) f_k}_{\text{how } g_j \text{ will change if } u_{jk} \text{ is higher/lower}}$
(ii) update the weight
$u_{jk} \leftarrow u_{jk} - \eta \frac{\partial E}{\partial u_{jk}}$
Copyright © 2014 Victor Lavrenko


Backpropagating dE/dy
y_j j z_j y_i i
$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$
$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$
$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$



import

nump

y as

np

import

decim

al

import

math

np.set_printoptions(prec

ision=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

b

2

=

0

.

5

x

1

=

0

x

2

=

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+ma
th.exp(-
zin1))
z1=round(z
1,4)
z2=1/(1+ma
th.exp(-
zin2))
z2=round(z
2,4)
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("calculat
e net output")
y=1/(1+math.
exp(-yin))
print("y=",y)
```

f

y

i

n

=

y

*

(

1

-

y

)

d

k

=

(

1

-

y

)

*

f

y

i

n

```
p
r
i
n
t
(
"
d
k
"
,
d
k
)

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
=",di
n1)
print(
"din2
=",di
n2)

print("e
rror in
delta")
fzin1=
z1 *(1-
z1)
print("f
zin1",fz
in1)
d1=din
1*
fzin1
fzin2=
```

```
z2 *(1-
z2)
print("f
zin2",fz
in2)
d2=din
2*
fzin2

print("d1=",d1)
print("d2=",d2)

print("Changes in weights between input and
hidden layer")dv11=alpha * d1 * x1
print("d
v11=",d
v11)
dv21=al
pha *
d1 * x2
print("d
v21=",d
v21)
dv01=al
pha *
d1
print("d
v01=",d
```

```
v01)
dv12=al
pha *
d2 * x1
print("d
v12=",d
v12)
dv22=al
pha *
d2 * x2
print("d
v22=",d
v22)
dv02=al
pha *
d2
print("d
v02=",d
v02)

print("Final weights
of network")
v1[0]=v1[0]+dv11
v1[1]=v1[1]+dv12
print("v=",v1)

v2[0]
=v2[
0]+d
```

```
v21
v2[1]
=v2[
1]+d
v22
print("v2",v2)

w[1]=w[1]+dw1
w[
2]
=w
[2]
+d
w2
b1
=b
1+
dv
01
b2
=b
2+
dv
02
w[
0]
=w
[0]
```

+d

w0

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

i
m
p
o
r
t

m
a
t

h

a
0
=
-
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.t
anh(n1)
n2=float(
w20*a1+b
20)
a2=math.t
anh(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=b
20-
(c*s2)
b11=b
10-
(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
```

P

ractical 5a:Aim: Write a program for Hopfield Network.

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.

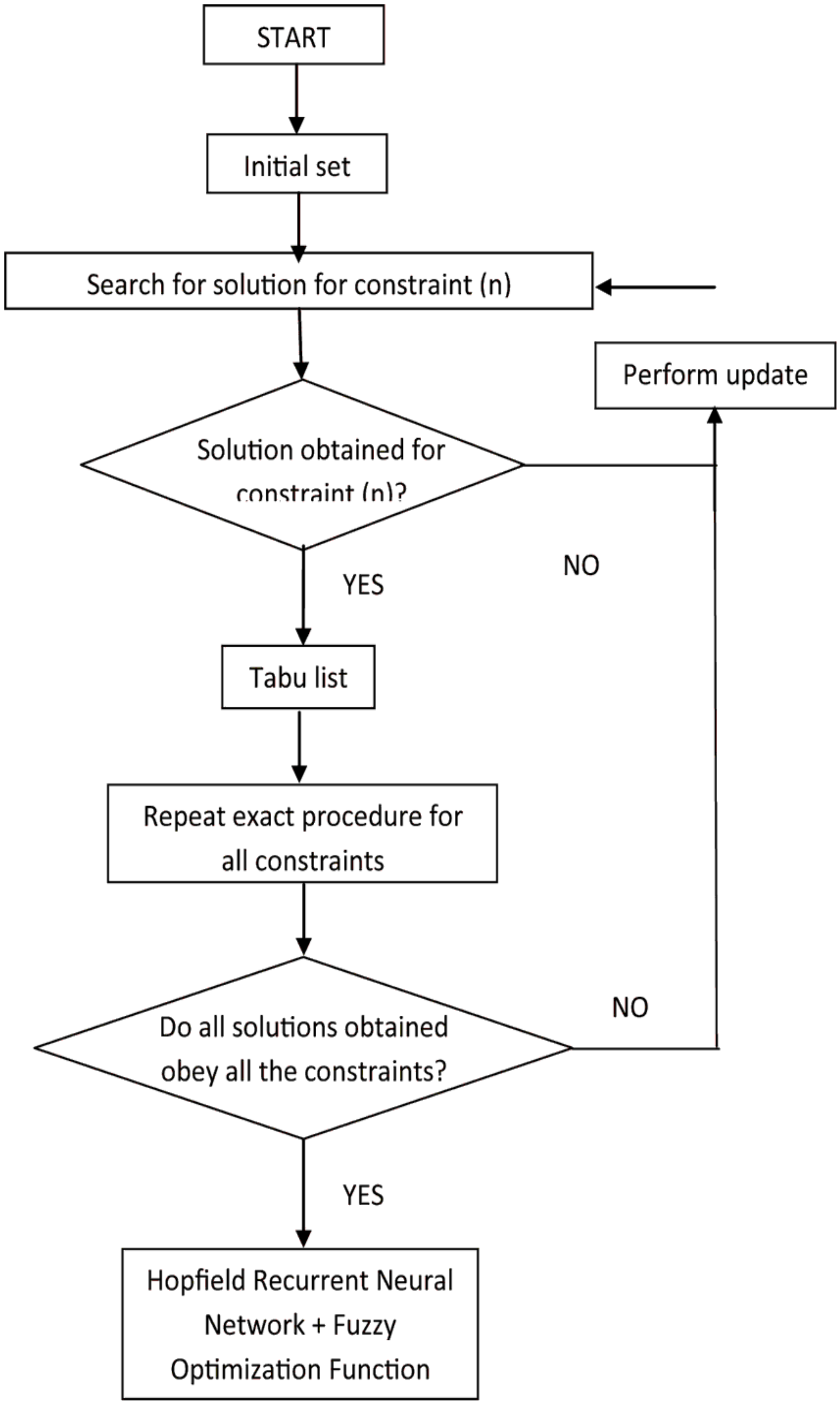
Step 4. Change activity on selected unit:

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

Step 5. Apply output function:

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

Step 6. Check stopping condition.


START
Initial set
Search for solution for constraint (n)
Perform update
Solution obtained for constraint (n)?
NO
YES
Tabu list
Repeat exact procedure for all constraints
NO
Do all solutions obtained obey all the constraints?
YES
Hopfield Recurrent Neural Network + Fuzzy Optimization Function

```
#include "hop.h"
neuron::neuron(i
nt *j)
{
int
i;
fo
r(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)
    r
e
t
u
r
n
```

```
(
1
)
;

e
l
s
e
    return (0);
}
network::network(int a[4],int b[4],int c[4],int d[4])
{
nrn[0]
=
neuro
n(a)   ;
nrn[1]
=
neuro
n(b)   ;
nrn[2]
=
neuro
n(c)   ;
nrn[3]
=
neuro
n(d) ;
}



void network::activation(int *patrn)
{
int
i,j;
fo
r(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].activ
    ation);

    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 LAYEROF";
cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE
NETWORK SHOULDRECALL 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 messagefor(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.activat
ion(patrn
2);
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
<iostrea
m.h>
#include
<math.h
> class
neuron
```

```
{
protected:
    int activation;
    friend
class
network;
public:
    int
    wei
    ght
    v[4
    ];
    neu
    ron
    ()
    {};
    neu
    ron
    (int
    *j)
    ;
    int
    act
    (int
    ,
    int
    *);
};
class network
{
public:
    ne
    ur
    on
    nr
    n[
    4];
    int
    ou
    tp
    ut[
    4];
    int
    thr
    es
```

```
    hl
    d(i
    nt)
    ;
    void activation(int
    j[4]);
    network(int*,int*,int
    *,int*);
};
```

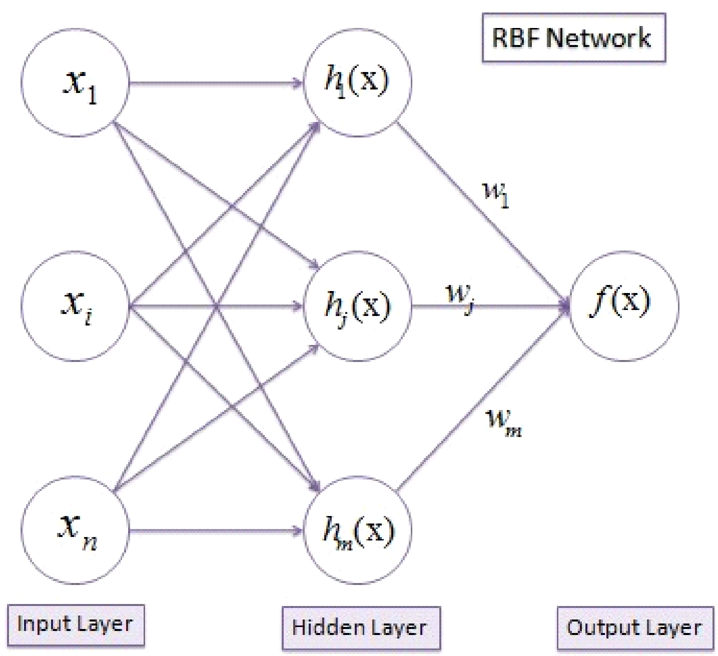
P

ractical 5b:Aim:Write a program for

Radial Basis function

Radial Basis Function Networks (RBF)

RBF networks have three layers: input layer, hidden layer and output layer. One neuron in the input layer corresponds to each predictor variable. With respects to categorical variables, *n*-1 neurons are used where *n* is the number of categories. Hidden layer has a variable number of neurons. Each neuronconsists of a radial basis function centered on a point with the same dimensions as the predictor variables. The output layer has a weighted sum of outputs from the hidden layer to form the network outputs.



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

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

Algorithm

$h(x)$ is the Gaussian activation function with the parameters r (the radius or standard deviation) and c (the center or average taken from the input space) defined separately at each RBF unit. The
learning process is based on adjusting the parameters of the network to reproduce a set of input-outputpatterns. There are three types of parameters; the weight w between the hidden nodes and the output nodes, the center c of each neuron of the hidden layer and the unit width r.

Unit Center (*c*)

Any clustering algorithm can be used to determine the RBF unit centers (e.g., K-means clustering). A
set of clusters each with r-dimensional centers is determined by the number of input variables or nodesof the input layer. The cluster centers become the centers of the RBF units. The number of clusters, H,

is a design parameter and determines the number of nodes in the hidden layer. The K-means clusteringalgorithm proceeds as follows:

- Initialize the center of each cluster to a different randomly selected training pattern.
- Assign each training pattern to the nearest cluster. This can be accomplished by calculating theEuclidean distances between the training patterns and the cluster centers.
- When all training patterns are assigned, calculate the average position for each cluster center.They then become new cluster centers.
- Repeat steps 2 and 3, until the cluster centers do not change during the subsequent iterations.

Unit width (*r*)

When the RBF centers have been established, the width of each RBF unit can be calculated using the K-nearest neighbors algorithm. A number K is chosen, and for each center, the K nearest centers is found. The root-mean squared distance between the current cluster center and its K nearest neighbors is calculated, and this is the value chosen for the unit width (r). So, if the current cluster center is c_j,
the r value is:

$$r_j = \sqrt{\frac{\sum_{i=1}^{k} (c_j - c_i)^2}{k}}$$

A typical value for K is 2, in which case s is set to be the average distance from the two nearestneighboring cluster centers.

Weights (*w*)

Using the linear mapping, **w** vector is calculated using the output vector (**y**) and the design matrix **H**.

$$y = wH$$
$$w = (H'H)\, H'y$$

The basis functions are (unnormalized) gaussians, the output layer is linear and the weightsare learned by a simple pseudo-inverse.

```
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(s
    elf, c, d):
    assert len(d)
    ==self.indi
    m
    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.numCe
    nters]self.centers =[X[i,:] for i in rnd_idx]
```

```
        print("center", self.centers)
        # calculate
        activations of
        RBFsG
        =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._
        calcAc
        t(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)

r

b

f

.

t

r

a

i

n

(

x

,

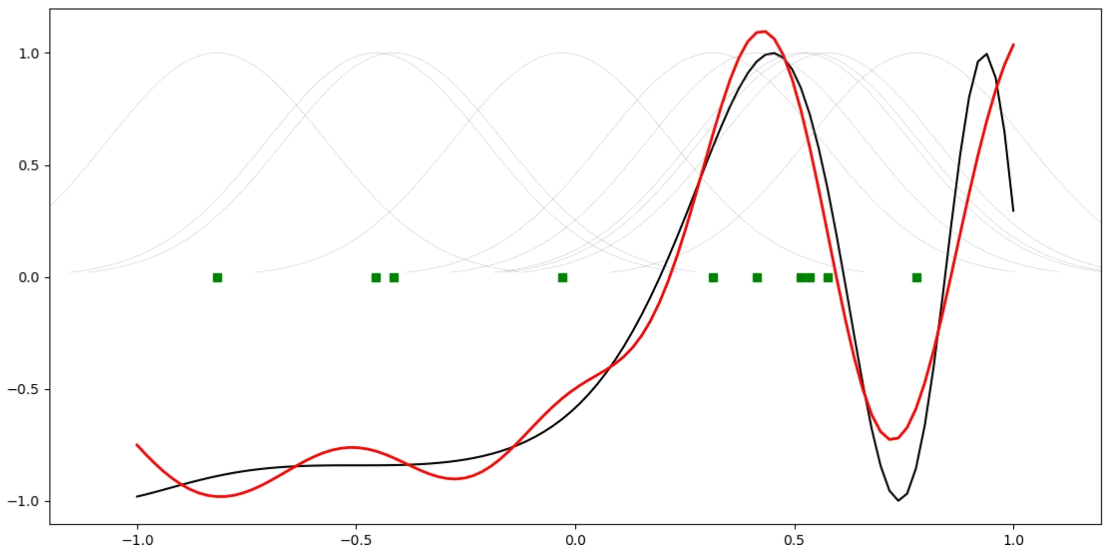
y

)

z

```
=
r
b
f
.
t
e
s
t
(
x
)

# plot original
data
plt.figure(figsi
ze=(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.x
lim(
-1.2,
1.2)
plt.s
how
()
```



Practical 6a:

Aim:Self-Organizing Maps

The SOM algorithm is used to compress the information to produce a similarity graph while preserving the topologic relationship of the input data space.

The basic SOM model construction algorithm can be interpreted as follows:

• Create and initialize a matrix (weight vector) randomly to hold the neurons. If the matrix can be initialized with order and roughly compiles with the input density function, the map willconverge quickly

• Read the input data space. For each observation (instance), use the optimum fit approach, which is based on the Euclidean distance

c = arg_i min || x – m_i ||

to find the neuron which best matches this observation. Let x denote the training vector from the observation and mi denote a single neuron in the matrix. Update that neuron to resemble that observation using the following equation:

$mi\ (t + 1) = mi\ (t) + h(t)[x(t) - mi\ (t)]$ (4) $mi(t)$:

the weight vector before the neuron is updated.

$(t + 1)$: the weight vector after the

neuron is updated.(t): the training

vector from the observation.

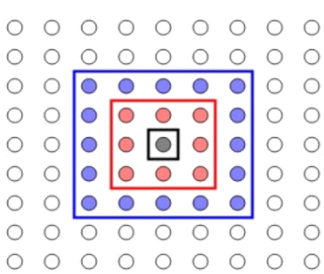
$h(t)$: the neighborhood function (a smoothing kernel defined over the lattice points), definedthough the following equation:

$h(t)$ = { $\alpha(t)$, $i \in Nc$ 0, $i \in Nc$ (5)

: the neighborhood set, which decreases with time.

(t): the learning-rate factor which can be linear, exponential or inversely

proportional.It is a monotonically decreasing function of time (t)



In general, SOMs might be useful for visualizing high-dimensional data in terms of its similarity structure. Especially large SOMs (i.e. with large number of Kohonen units) are known to perform mappings that preserve the topology of the original data, i.e. neighboring data points in input space will also be represented in adjacent locations on the SOM.

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],

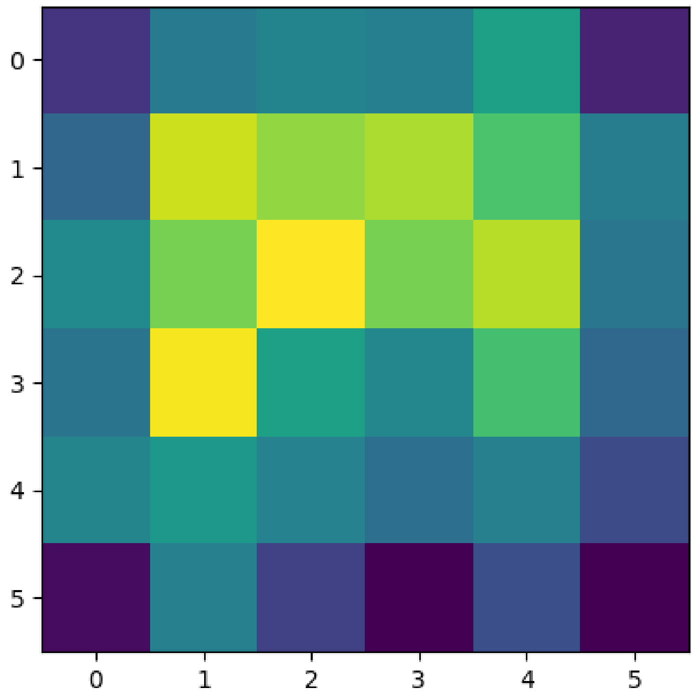
[0.77, 0.59, 0.22, 0.03]]

som = MiniSom(6, 6, 4, sigma=0.3, learning_rate=0.5) #

initialization of 6x6 SOMsom.train_random(data, 100) # trains the

SOM with 100 iterations plt.imshow(som.distance_map())

plt.show()



Practical 7a:

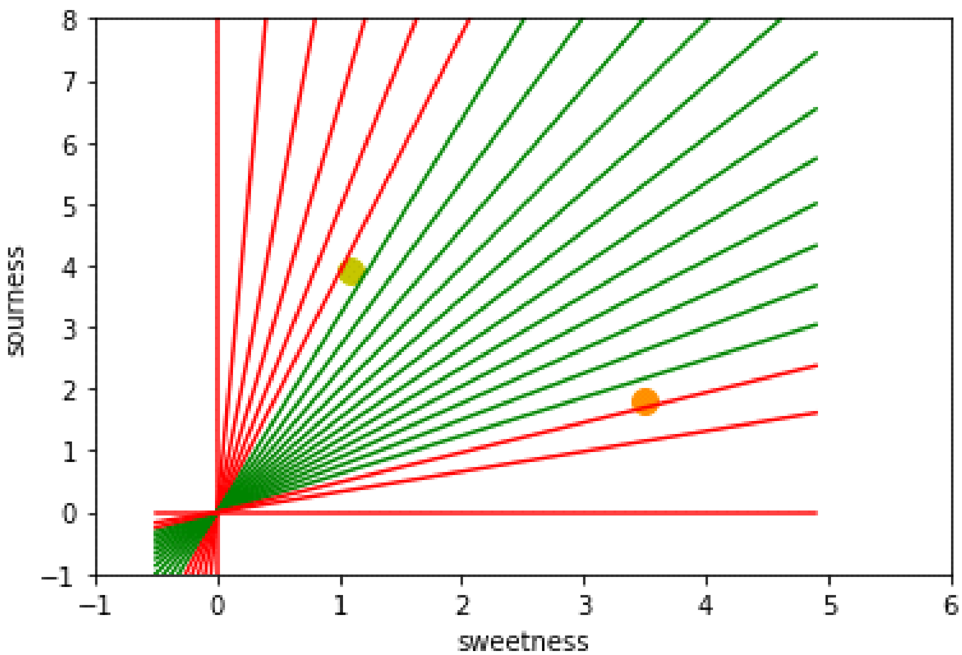
Aim: Line Separation

You could imagine that you have two attributes describing am eddible object like a fruit for example: "sweetness" and "sourness"
We could describe this by points in a two-dimensional space. The x axis for the sweetness and the y axis for the sourness. Imagine now that we have two fruits as points in this space, i.e. an orange at position (3.5, 1.8) and a lemon at (1.1, 3.9).
We could define dividing lines to define the points which are more lemon-like and which are more orange-like. The following program calculates and renders a bunch of lines. The red ones are completely unusable for this purpose, because they are not separating the classes. Yet, it is obvious that even the green ones are not all useful.

```
import numpy as np
import matplotlib.pyplot as plt
def
   create_distance_func
   tion(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
    + cif 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 samplessize
= 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.pl

ot(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 Si=+1 and 'OFF' (silent) if Si=−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 Δt=1ms, we can interpret Si(t)=+1 as an action potential of neuron i at time t. If we take Δt=500ms, Si(t)=+1 should rather be interpreted as an episode of highfiring rate.

Neurons interact with each other with weights wij. The input potential of neuron i, influencedby the activity of other neurons is

$$hi(t)=\sum jwijSj(t). \qquad (17.2)$$

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

$$Prob\{Si(t+\Delta t)=+1|hi(t)\}=g(hi(t))=g(\sum jwijSj(t)) \qquad (17.3)$$

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

$$S_i(t+\Delta t)=\mathrm{sgn}[h(t)] \qquad (17.4)$$

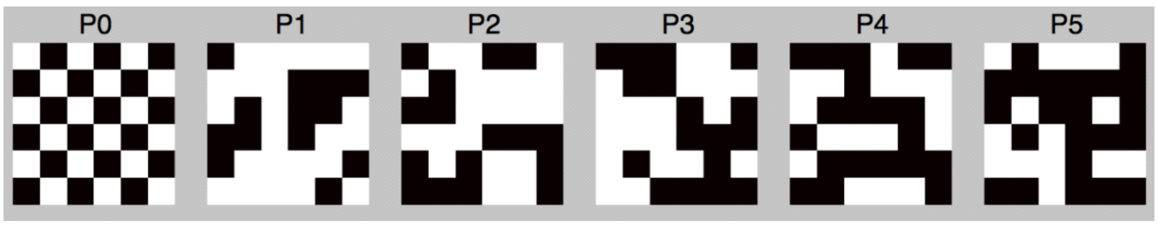
For finite β the dynamics are stochastic. In the following we assume that in each time step all neurons are updated synchronously (parallel dynamics), but an update scheme where only oneneuron is updated per time step is also possible.

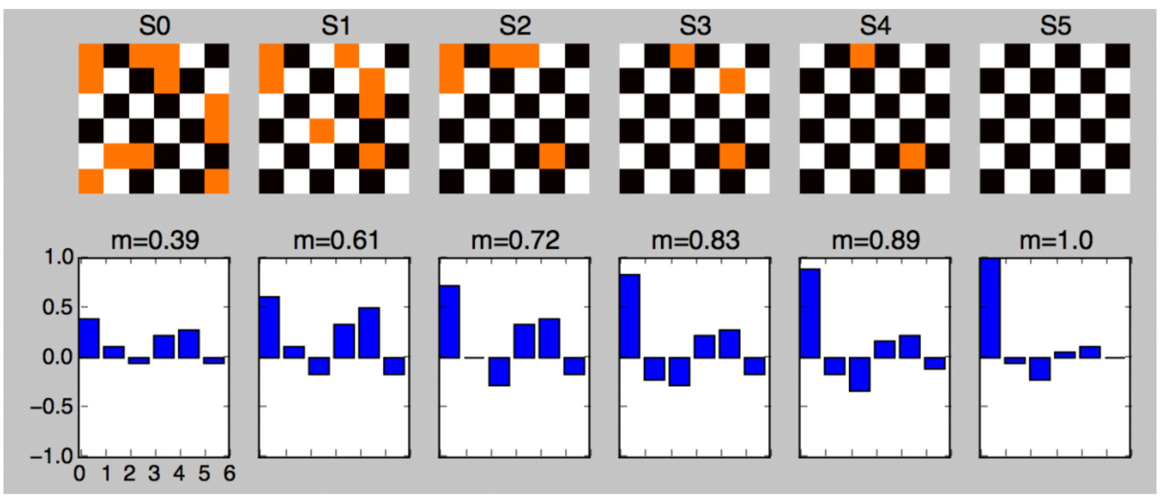
Source code:

```
%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 listcheckerboard =
    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
    overlapsoverlap_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 networknoisy_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_step
s=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")





P

ractical 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 listsdef
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
        r
et
ur
n
0
lis
t1
=[
1,
2,
3,
4,
5]
list2=[6,7,8,9]
if(overlapping
        (list1,l
        ist2)):
        print("
        overla
        pping"
        )
else:
        print("not overlapping")
```

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

Define a function() that
takes two listsdef
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
r
et
ur
n
0
lis
t1
=[
1,
2,
3,
4,
5]
list2=[6,7,8,9]

```
if(overlapping
        (list1,l
        ist2)):
        print("
        overla
        pping"
        )
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)
```

P

ractical 9b:Aim: Solve Tipping

Problem using fuzzy logic

Fuzzy Control Systems: The Tipping Problem
==

The 'tipping problem' is commonly used to illustrate the power of fuzzy logicprinciples to generate complex behavior from a compact, intuitive set of expert rules.

If you're new to the world of fuzzy control systems, you might wantto check out the `Fuzzy Control Primer <../userguide/fuzzy_control_primer.html>`_ before reading through this worked example.

The Tipping Problem

Let's create a fuzzy control system which models how you might choose to tipat a restaurant. When tipping, you consider the service and food quality,
rated between 0 and 10. You use this to leave a tip of between

0 and 25%.We would formulate this problem as:

- Antecednets (Inputs)
  - `service`
    - Universe (ie, crisp value range): How good was the service of the waitstaff, on a scale of 0 to 10?
    - Fuzzy set (ie, fuzzy value range): poor, acceptable, amazing
  - `food quality`
    - Universe: How tasty was the food, on a scale of 0 to 10?
    - Fuzzy set: bad, decent, great
- Consequents (Outputs)

- `tip`
  - Universe: How much should we tip, on a scale of 0% to 25%
  - Fuzzy set: low, medium, high
- Rules
  - IF the *service* was good *or* the *food quality* was good,THEN the tip will be high.
  - IF the *service* was average, THEN the tip will be medium.
  - IF the *service* was poor *and* the *food quality* was poorTHEN the tip will be low.
- Usage
  - If I tell this controller that I rated:
    - the service as 9.8, and
    - the quality as 6.5,
  - it would recommend I leave:
    * a 20.2% tip.

Creating the Tipping Controller Using the skfuzzy control API

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

Code:

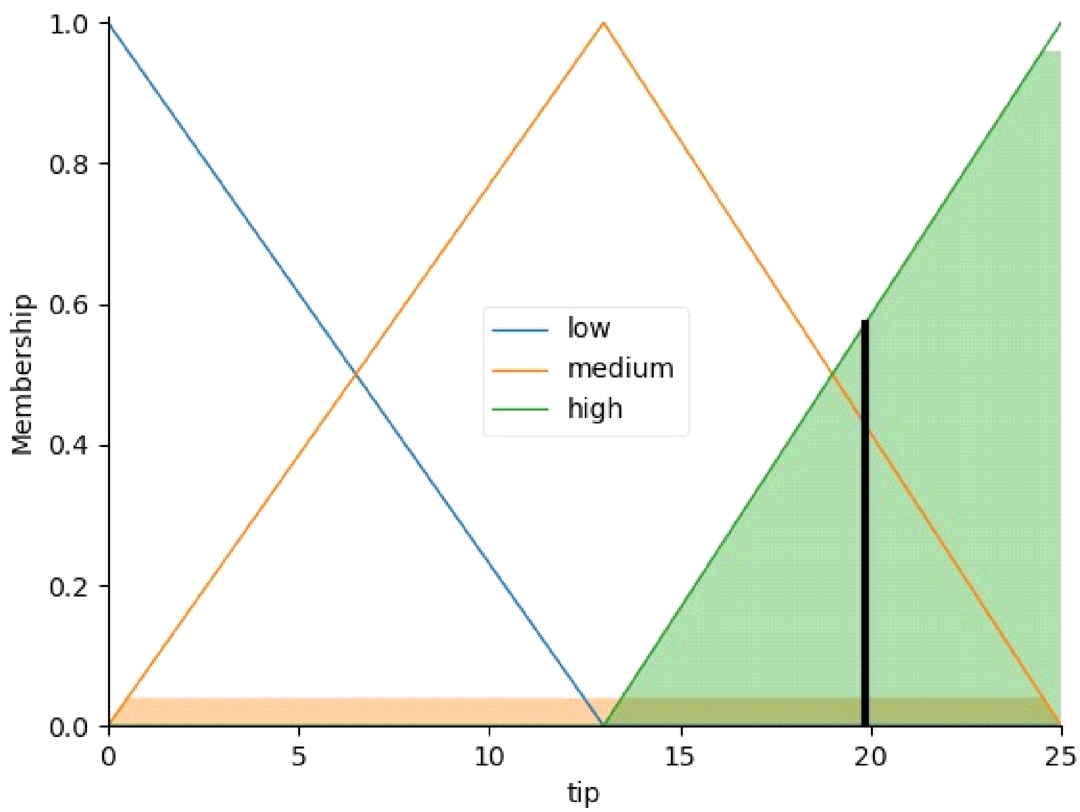
```
import
numpy as
np import
skfuzzy as
fuzz
from skfuzzy import control as ctrl

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')

qual
ity.
auto
mf(
3)
serv
ice.
auto
mf(
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()

s
e
r
v
i
c
e
.
v
i
e
w
(
)
t
i
p
.
v
i
e
w
(
)


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 APItipping.input['quality'] = 6.5
tipping.input['service'] = 9.8

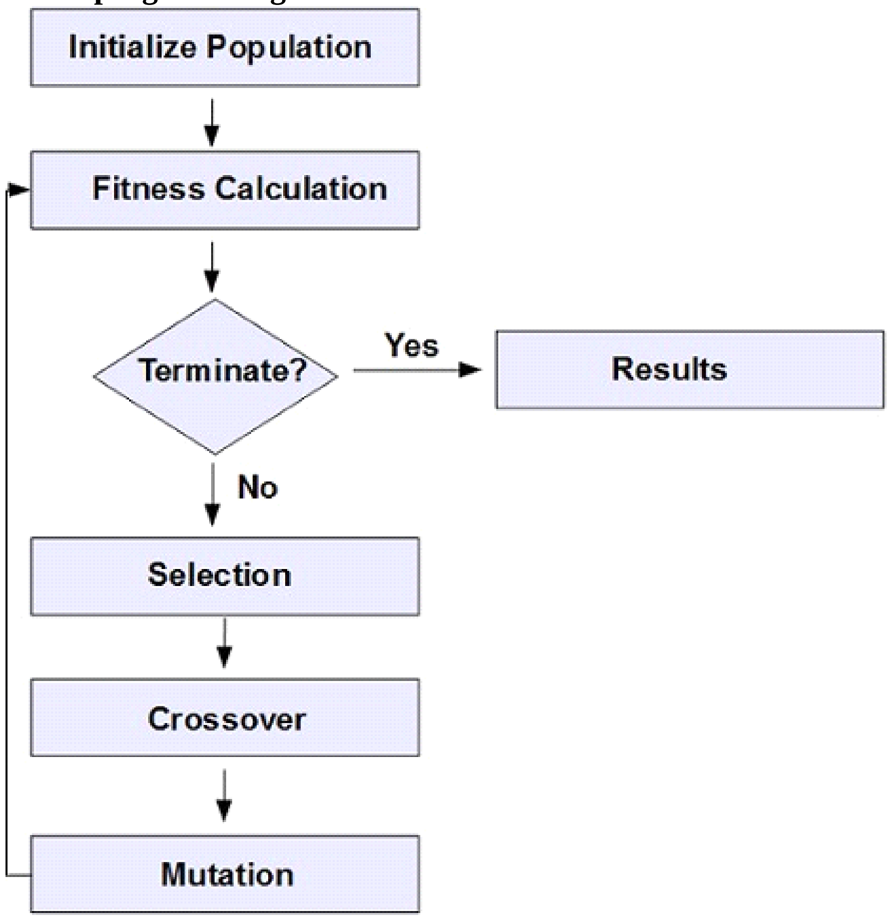
#
Crunch
the
numbers
tipping.c
ompute(
)

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

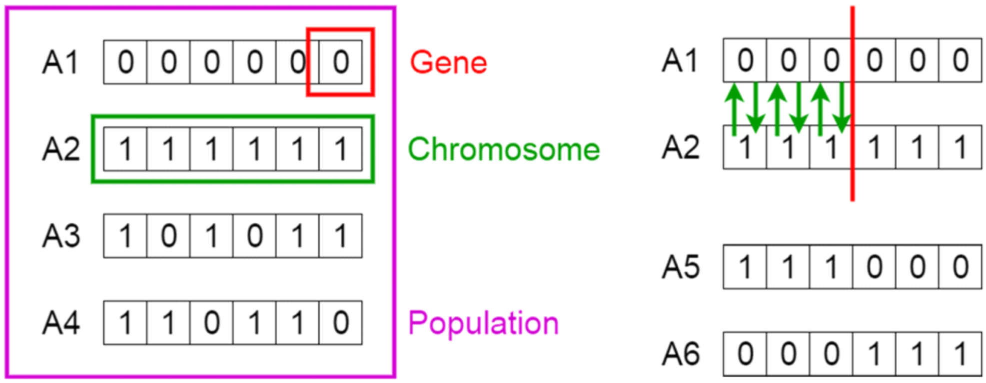


The resulting suggested tip is ** 19.8476**

P
ractical 10:Aim: Implementation of simple genetic algorithm



Genetic Algorithms




```
import random

# Number of individuals in
each generation
POPULATION_SIZE =100

# Valid genes
GENES
='''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLM
NOPQRSTUVWXYZ 1234567890, .-
;:_!"#%&/()=?@${[]}'''

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

class
   Individ
   ual(obje
   ct):'''
   Class representing individual
   in population'''
   def__init_(self,
      chromosome):
      self.chromosome
```

```
    =chromosome
    self.fitness
    =self.cal_fitness()

@classmethod
def
    mutated
    _genes(s
    elf):'''
    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(TARGE
    T)
    return[self.mutated_genes() for _ in range(gnome_len)]

def
    mat
    e(se
    lf,
    par2
    ):'''
    Perform mating and produce
    new offspring'''

    # chromosome
    for offspring
    child_chromos
    ome =[]
    for gp1, gp2 in zip(self.chromosome, par2.chromosome):
```

```
        # random
        probability
        prob
        =random.ra
        ndom()

        # 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(gp
            2)

        # 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_chromos
    ome)

def
    cal_
    fitne
    ss(s
    elf):
    '''
    Calculate fittness score, it is
    the number of characters in
    string which differ from target
    string.
    '''
    glo
    bal
```

```
        TA
        R
        GE
        T
        fit
        nes
        s
        =0

        for gs, gt in
           zip(self.chromosome,
           TARGET):if gs !=gt:
           fitness+=1
        return fitness

# 

D
r
i
v
e
r

c
o
d
e

d
e
f

m
a
i
n
(
)
:
    global POPULATION_SIZE

    #curr
    ent
    gener
    ation
```

gener
ation
=1

f
o
u
n
d
=
F
a
l
s
e
p
o
p
u
l
a
t
i
o
n
=
[
]

create initial population
for _ in
range(POPULATION_
SIZE): gnome
=Individual.create_gno
me()
population.append(Individua

l(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 generationnew_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(chi
      ld)

    population =new_generation

    print("Generation: {}\tString: {}\tFitness:
{}".format(generation,"".join(population[0].chromosome),populati
    on[0].fitness))generation +=1


  print("Generation: {}\tString: {}\tFitness:
{}".format(generation,
"".join(population[0].chromosome),
     population[0].fitness))
```

if__name__=='_main_':main()

Output:

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 pltfrom tkinter import Tk, Canvas, Frame,

BOTH, Text

i

m

p

o

r

t

m

a

t

h

c

l

a

```
s
s

C
i
t
y
:
    def__init
        (self, x,
        y):self.x
        = x
        self.y = y

    def
        distance(sel
        f,  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
    routeDi
    stance(s
    elf):if
    self.dist
    ance
    ==0:

        pathDistance = 0
        for i in range(0,
            len(self.route)):
            fromCity =
            self.route[i]
            toCity = None
            if i + 1 <
                len(self.rou
                te): toCity
                =
                self.route[i
                + 1]
```

```
                else:
                    toCity = self.route[0]
                pathDistance +=
            fromCity.distance(toCity)
            self.distance = pathDistance
        return self.distance

    def
        route
        Fitnes
        s(self)
        :if
        self.fi
        tness
        == 0:
            self.fitness = 1 /
        float(self.routeDistance())
        return self.fitness
def createRoute(cityList):
    route = random.sample(cityList,
    len(cityList))return route
def
    initialPopulation(popSi
    ze, cityList):population
    = []
    for i in range(0, popSize):
        population.append(createRoute(cityList))
    return population
```

```
def
    rankRoutes(
    population):
    fitnessResul
    ts = {}
    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(popRanke
    d, 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(popR
                anked[i][0])break
    return selectionResults



def matingPool(population,
    selectionResults):matingpool
    = []
    for i in range(0,
        len(selectionResults)):
        index =
        selectionResults[i]
        matingpool.append(popul
        ation[index])
    return matingpool


def breed(parent1, parent2):

    c
    h
    i
    l
    d
```

=

[

]

c

h

i

l

d

P

1

=

[

]

c

h

i

l

d

P

2

=

```
[
]

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(matingpoo
    l, eliteSize):children = []
    length = len(matingpool) - eliteSize
    pool = random.sample(matingpool, len(matingpool))

    for i in
        range(0,eliteSize):
        children.append(m
        atingpool[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[swap
            ped] city2 =
```

```
            individual[swap
            With]

            individual[swap
            ped] = city2
            individual[swap
            With] = 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(progre
    ss)
    plt.ylabel('Dist
    ance')
    plt.xlabel('Gen
    eration')
    plt.show()
def
    m
    a
    i
    n
    (
    )
    :
    c
    i
    t
    y
    L
    i
    s
    t
    =

    [
    ]
    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()
```