INDEX

Practical No:01

Aim: Design a simple linear neural network model.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
float x,b,w,net;
float out;
cout<<"\n Enter the input X:";
cin>>x;
cout<<"\n Enter the bias b:";
cin>>b;
cout<<"\n Enter the weight W:";
cin>>w;
net=(w*x+b);
cout<<"\n ______________________OUTPUT__________________ ";
cout<<"\n net+ "<<net<<endl;
if(net<0)
{ out=0;
}
else
if((net>=0)&&(net<=1))
{
out=net;
}
else
out=1;
cout<<"OUTPUT:"<<out<<endl;
getch();
}
```

```
 Enter the input X:1

 Enter the bias b:1

 Enter the weight W:3

 ____________________OUTPUT________________
 net+ 4
OUTPUT:1

_
```

Sign:_____________

Practical No:02

Aim: Calculate the output of the neural net using both binary and sigmoidal function.

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int i;
 float inp[3] = {0.3,0.5,0.6};
 float wght[3] ={0.2,0.1,-0.3};
 float yin=0;

 cout<<"The inputs are: \n";
 for(i=0;i<3;i++)
 {
  cout<<inp[i];
  cout<<"\n";
  }


  cout<<"************************************************* \n";
 cout<<"The weights are: \n";
 for(i=0;i<3;i++)
 {
  cout<<wght[i];
  cout<<"\n";
  }
    cout<<"******************************* \n";
    cout<<"\n the net input can be calculated as Yin= x1w1+.....+XnWn. \n";

    for(i=0;i<3;i++)
    {
     yin= yin+inp[i]*wght[i];
     }
     cout<<yin;


  getch();
  }
```

```
The inputs are:
0.3
0.5
0.6
*********************************************
The weights are:
0.2
0.1
-0.3
*******************************

 the net input can be calculated as Yin= x1w1+.....+XnWn.
-0.07_
```

Sign:_________________

Practical No: 03

Aim: Generate AND/NOT function using McCulloch-Pitts neural net.

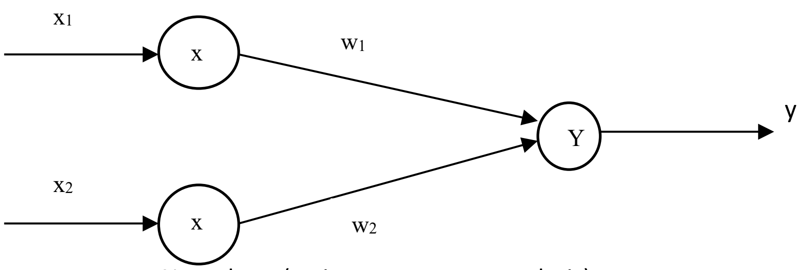
Solution:

In the case of ANDNOT 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 be decided only after the analysis. The net can be represent as shown in figure below:



Neural net (wei... er 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 x 1 + l x 1 = 2

(1, 0), y_{ij} = 1 x 1 + 0 x 1 = 1

(0, 1), y_{ij} = 0 x 1 + 1 x 1 = 1

(0, 0), y_{ij} = 0 x 1 + 0 x 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 by fixing a threshold of 1, i.e., $\vartheta \geq 1$ for Y unit. Thus,

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

Note: The value is calculated using the following:

$\vartheta \geq nw - p$

$\vartheta \geq 2 \times 1 - 1$

$\vartheta \geq 1$

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:

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
import numpy
# 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)
```

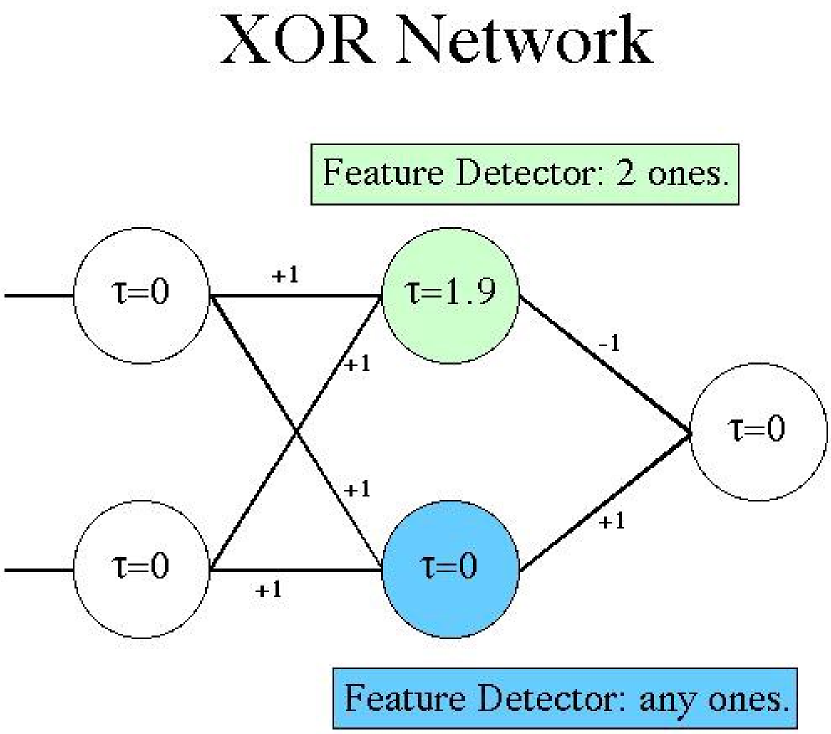
```
print("x1 = ",x1)
print("x2 = ",x2)

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(n1[i] - m1[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):
    ele =  1
    Y.append(ele)
  if(Yin[i]<1):
    ele =  0
    Y.append(ele)
print("Y = ",Y)
```

```
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]: 
```

Sign:__________________

Practical No:04

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




```
import numpy
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt

# The following code is used for hiding the warnings and make this notebook clearer.
import warnings
warnings.filterwarnings('ignore')
def tanh(x):
   return (1.0 - numpy.exp(-2*x))/(1.0 + numpy.exp(-2*x))

def tanh_derivative(x):
   return (1 + x)*(1 - x)
class NeuralNetwork:
   #########
   # parameters
   # ----------
   # self:      the class object itself
   # net_arch:  consists of a list of integers, indicating
   #            the number of neurons in each layer, i.e. the network architecture
   #########
```

```
    def __init__(self, net_arch):
        numpy.random.seed(0)

        # Initialized the weights, making sure we also
        # initialize the weights for the biases that we will add later
        self.activity = tanh
        self.activity_derivative = tanh_derivative
        self.layers = len(net_arch)
        self.steps_per_epoch = 1
        self.arch = net_arch
        self.weights = []

        # Random initialization with range of weight values (-1,1)
        for layer in range(self.layers - 1):
            w = 2*numpy.random.rand(net_arch[layer] + 1, net_arch[layer+1]) - 1
            self.weights.append(w)

    def _forward_prop(self, x):
        y = x

        for i in range(len(self.weights)-1):
            activation = numpy.dot(y[i], self.weights[i])
            activity = self.activity(activation)

            # add the bias for the next layer
            activity = numpy.concatenate((numpy.ones(1), numpy.array(activity)))
            y.append(activity)

        # last layer
        activation = numpy.dot(y[-1], self.weights[-1])
        activity = self.activity(activation)
        y.append(activity)

        return y

    def _back_prop(self, y, target, learning_rate):
        error = target - y[-1]
        delta_vec = [error * self.activity_derivative(y[-1])]

        # we need to begin from the back, from the next to last layer
        for i in range(self.layers-2, 0, -1):
            error = delta_vec[-1].dot(self.weights[i][1:].T)
            error = error*self.activity_derivative(y[i][1:])
            delta_vec.append(error)

        # Now we need to set the values from back to front
        delta_vec.reverse()

        # Finally, we adjust the weights, using the backpropagation rules
        for i in range(len(self.weights)):
            layer = y[i].reshape(1, self.arch[i]+1)
```

```
            delta = delta_vec[i].reshape(1, self.arch[i+1])
            self.weights[i] += learning_rate*layer.T.dot(delta)

    #########
    # parameters
    # ----------
    # self:    the class object itself
    # data:    the set of all possible pairs of booleans True or False indicated by the integers 1 or 0
    # labels:  the result of the logical operation 'xor' on each of those input pairs
    #########
    def fit(self, data, labels, learning_rate=0.1, epochs=100):

        # Add bias units to the input layer -
        # add a "1" to the input data (the always-on bias neuron)
        ones = numpy.ones((1, data.shape[0]))
        Z = numpy.concatenate((ones.T, data), axis=1)

        for k in range(epochs):
            if (k+1) % 10000 == 0:
                print('epochs: {}'.format(k+1))

            sample = numpy.random.randint(X.shape[0])

            # We will now go ahead and set up our feed-forward propagation:
            x = [Z[sample]]
            y = self._forward_prop(x)

            # Now we do our back-propagation of the error to adjust the weights:
            target = labels[sample]
            self._back_prop(y, target, learning_rate)

    #########
    # the predict function is used to check the prediction result of
    # this neural network.
    #
    # parameters
    # ----------
    # self:   the class object itself
    # x:      single input data
    #########
    def predict_single_data(self, x):
        val = numpy.concatenate((numpy.ones(1).T, numpy.array(x)))
        for i in range(0, len(self.weights)):
            val = self.activity(numpy.dot(val, self.weights[i]))
            val = numpy.concatenate((numpy.ones(1).T, numpy.array(val)))
        return val[1]

    #########
    # the predict function is used to check the prediction result of
    # this neural network.
    #
```

```
    # parameters
    # ----------
    # self:   the class object itself
    # X:      the input data array
    #########
    def predict(self, X):
        Y = numpy.array([]).reshape(0, self.arch[-1])
        for x in X:
            y = numpy.array([[self.predict_single_data(x)]])
            Y = numpy.vstack((Y,y))
        return Y
numpy.random.seed(0)

# Initialize the NeuralNetwork with
# 2 input neurons
# 2 hidden neurons
# 1 output neuron
nn = NeuralNetwork([2,2,1])

# Set the input data
X = numpy.array([[0, 0], [0, 1],
                [1, 0], [1, 1]])

# Set the labels, the correct results for the xor operation
y = numpy.array([0, 1,
                 1, 0])

# Call the fit function and train the network for a chosen number of epochs
nn.fit(X, y, epochs=100000)

# Show the prediction results
print("Final prediction")
for s in X:
    print(s, nn.predict_single_data(s))
```

```
epochs: 10000
epochs: 20000
epochs: 30000
epochs: 40000
epochs: 50000
epochs: 60000
epochs: 70000
epochs: 80000
epochs: 90000
epochs: 100000
Final prediction
[0 0] 2.769390318381638e-05
[0 1] 0.995154295769496
[1 0] 0.9951532983339081
[1 1] 2.7834280538588892e-05
```

Sign:__________________

Practical N0:05

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

```
#include<iostream>

using namespace std;

int main()

{

float wt,net,div,a,at,d,i,x,w,dw;

cout<<"consider a single neuron perceptron with a single i/p";

cin>>w;

 cout<<"enter the learning coefficient";

cin>>d;

 for ( i=0;i<10;i++)

{

net = x+w;

if(wt<0)

{

a=0;

 }

 else

{

   a=1;

div=at+a+w;

 w=w+div;

}

cout<<"i+1 in fraction are i"<<a<<"change in weight"<<dw<<"adjustment at="<<w;

}

}
```

options | compilation | execution

```
consider a single neuron perceptron with a single i/p2
enter the learning coefficient12
i+1 in fraction are i1change in weight0adjustment at=5i+1 in fraction are i1change in weight0adjustment at=11

Exit code: 0 (normal program termination)
```

Practical No: 06

Aim: Write a program to implement delta rule

```
#include<iostream.h>
 #include<conio.h>
 void main()
 {
clrscr( );
float input[3],d,weight[3],delta;
 for(int i=0;i < 3 ; i++)
{
cout<<”\n initilize weight vector “<<i<<”\t”;
cin>>input[i];
 }
cout<<””\n enter the desired output\t”;
cin>>d;
do {
del=d-a;
if(del<0)
for(i=0 ;i<3 ;i++)
w[i]=w[i]-input[i];
else
if(del>0)
for(i=0;i<3;i++)
weight[i]=weight[i]+input[i];
for(i=0;i<3;i++)
{
val[i]=del*input[i];
weight[+1]=weight[i]+val[i];
}
cout<<”\value of delta is “<<del;
cout<<”\n weight have been adjusted”;
}
while(del ≠ 0)
if(del=0)
cout<<”\n output is correct”;
}
```

Sign:__________________

Practical No: 07

Aim: Write a program for linear 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_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()
```

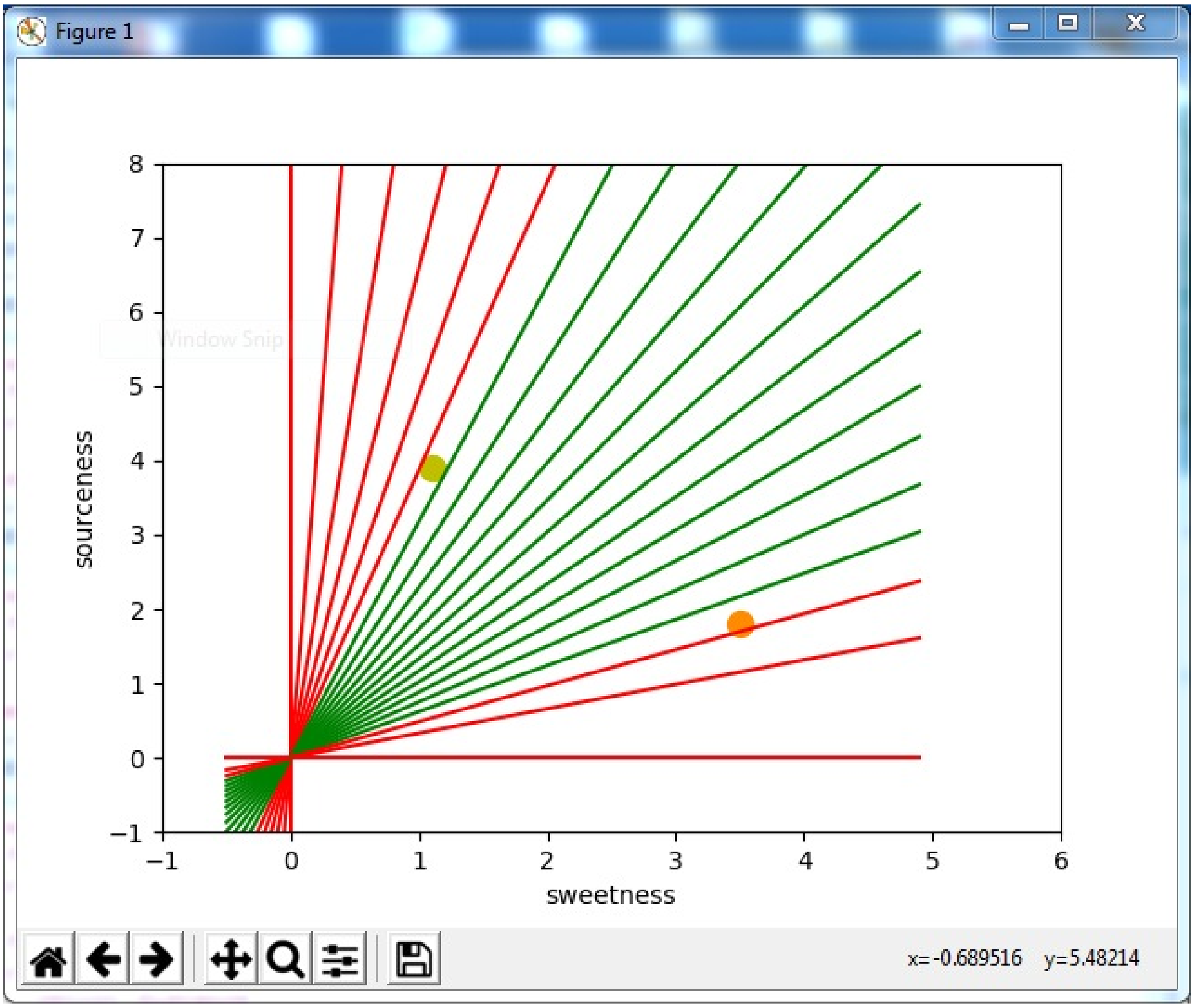


Figure 1
8
7
6
5
4
3
2
1
0
−1
sourceness
−1
0
1
2
3
4
5
6
sweetness
x=-0.689516 y=5.48214

Sign:________________

Practical No: 08

Aim: Write a program for Hopfield network model for 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 high firing rate.

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

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

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

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

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

$$Si(t+\Delta t)=\mathrm{sgn}[h(t)]$$

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.

The aim of this section is to show that, with a suitable choice of the coupling matrix wij memory items can be retrieved by the collective dynamics defined in Eq. (17.3), applied to all N neurons of the network. In order to illustrate how collective dynamics can lead to meaningful results, we start, in Section 17.2.1, with a detour through the physics of magnetic systems. In Section 17.2.2, the insights from magnetic systems are applied to the case at hand, i.e., memory recall.

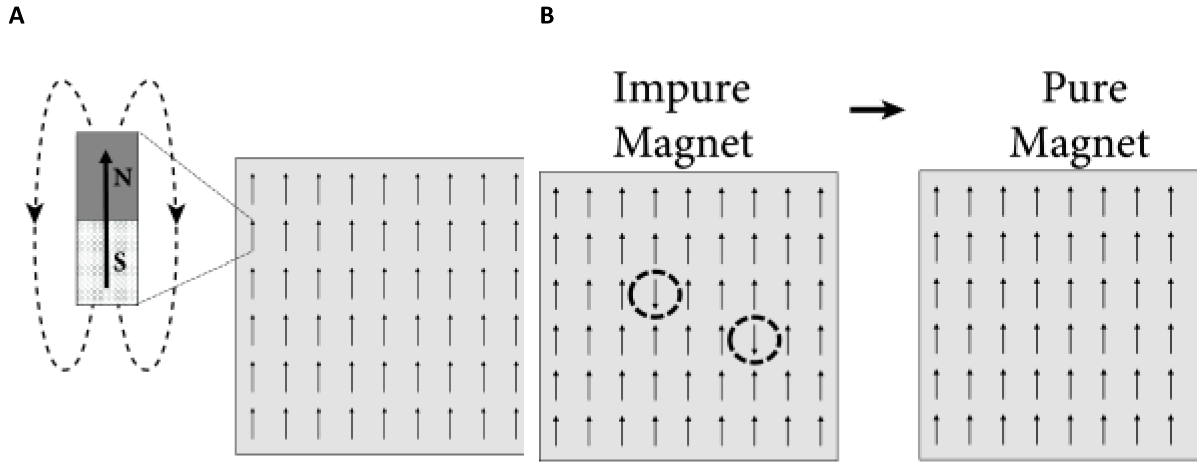
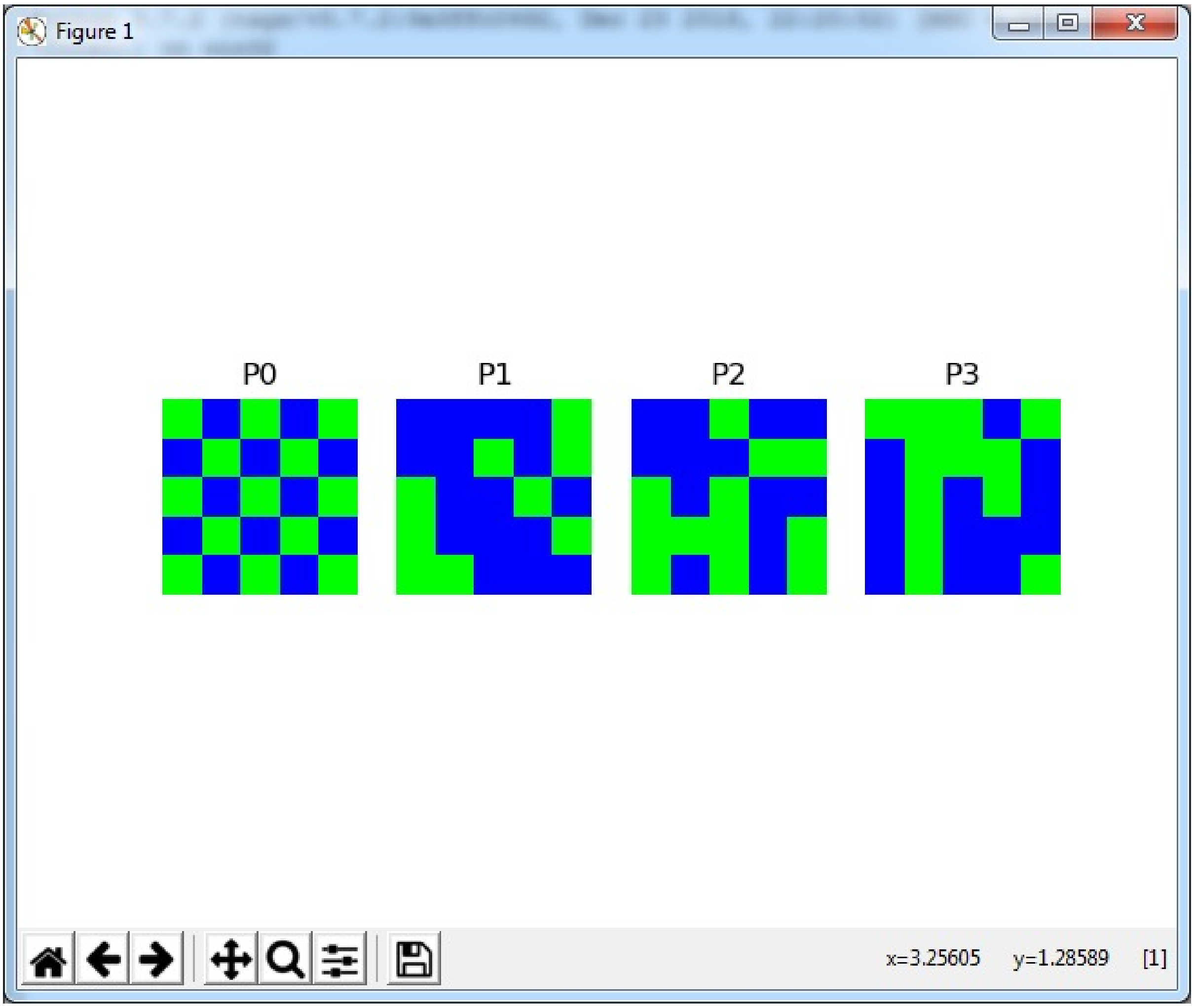


Fig. 17.5: Physics of ferromagnets. **A**. Magnetic materials consists of atoms, each with a small magnetic moment, here visualized as an arrow, a symbol for a magnetic needle. At low temperature, all magnetic needles are aligned. Inset: Field lines around one of the magnetic needles. **B**. At high temperature, some of the needles are misaligned (dashed circles). Cooling the magnet leads to a spontaneous alignment and reforms a pure magnet; schematic figure.

%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")
```



Sign:________________

Practical No: 09

Aim: Write a program to implement 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")
```

```
=== RESTART: C:/Users/yadne/AppData/Local/Programs/Python/Python38-32,
not overlapping
>>>
```

Sign:__________

```
# 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")
```

```
=== RESTART: C:/Users/yadne/AppData/Lo
not overlapping
>>>
```

Sign:__________________

Practical No: 10

Aim: Write a program to implement 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")
```

```
x = 5
if (type(x) is int):
	print ("true")
else:
	print ("false")
```

```
==========================================
true
>>>
>>>
```

```
# 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")
```

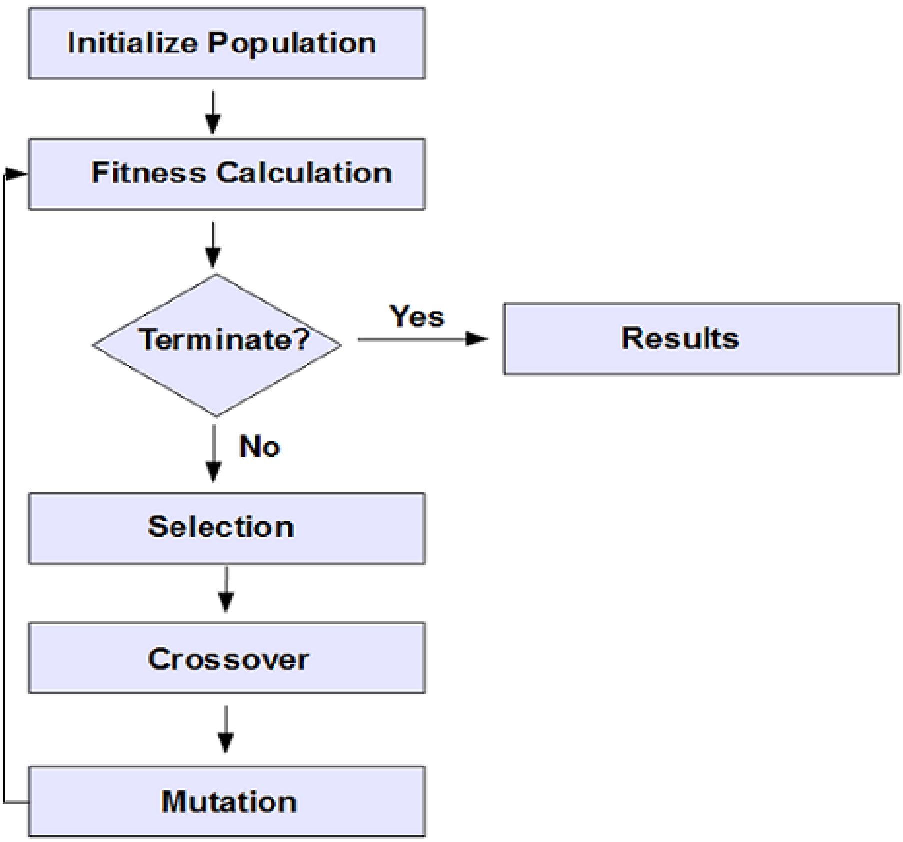
```
x = 5.2
if (type(x) is not int):
	print ("true")
else:
	print ("false")
```

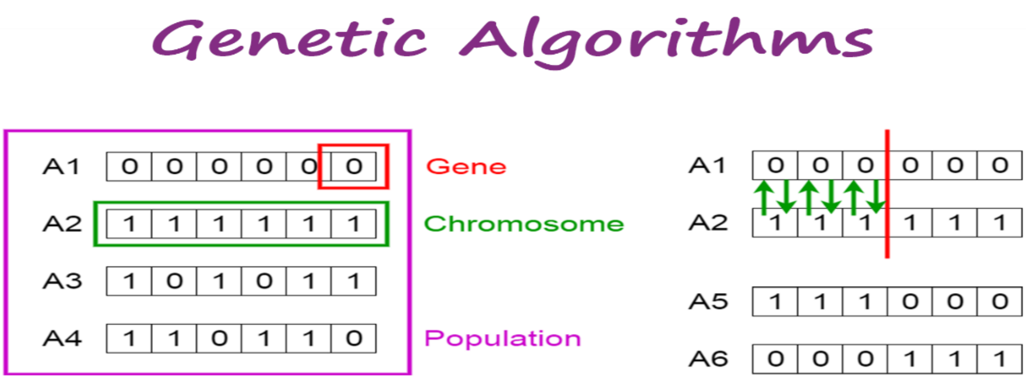
```
=========================
true
>>>
```

Sign:_________________

Practical No:11

Aim: Write a program to implement Simple genetic Algorithm.






```
import random

# Number of individuals in each generation
POPULATION_SIZE = 100

# Valid genes
GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP
```

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

# Target string to be generated
TARGET = "I love GeeksforGeeks"

class Individual(object):
    '''
    Class representing individual in population
    '''
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()

    @classmethod
    def mutated_genes(self):
        '''
        create random genes for mutation
        '''
        global GENES
        gene = random.choice(GENES)
        return gene

    @classmethod
    def create_gnome(self):
        '''
        create chromosome or string of genes
        '''
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]

    def mate(self, par2):
        '''
        Perform mating and produce new offspring
        '''

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

            # random probability
            prob = random.random()

            # if prob is less than 0.45, insert gene
            # from parent 1
            if prob < 0.45:
                child_chromosome.append(gp1)

            # if prob is between 0.45 and 0.90, insert
            # gene from parent 2
            elif prob < 0.90:
                child_chromosome.append(gp2)

            # otherwise insert random gene(mutate),
            # for maintaining diversity
            else:
                child_chromosome.append(self.mutated_genes())
```

```
        # create new Individual(offspring) using
        # generated chromosome for offspring
        return Individual(child_chromosome)

    def cal_fitness(self):
        '''
        Calculate fittness score, it is the number of
        characters in string which differ from target
        string.
        '''
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt: fitness+= 1
        return fitness

# Driver code
def main():
    global POPULATION_SIZE

    #current generation
    generation = 1

    found = False
    population = []

    # create initial population
    for _ in range(POPULATION_SIZE):
                gnome = Individual.create_gnome()
                population.append(Individual(gnome))

    while not found:

        # sort the population in increasing order of fitness score
        population = sorted(population, key = lambda x:x.fitness)

        # if the individual having lowest fitness score ie.
        # 0 then we know that we have reached to the target
        # and break the loop
        if population[0].fitness <= 0:
            found = True
            break

        # Otherwise generate new offsprings for new generation
        new_generation = []

        # Perform Elitism, that mean 10% of fittest population
        # goes to the next generation
        s = int((10*POPULATION_SIZE)/100)
        new_generation.extend(population[:s])

        # From 50% of fittest population, Individuals
        # will mate to produce offspring
        s = int((90*POPULATION_SIZE)/100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
```

```
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)

        population = new_generation

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

        generation += 1


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

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

```
================================================ RESTART: C:/User:
Generation: 1	String: 5#idvR pnw-LF@i]q@Q.	Fitness: 18
Generation: 2	String: 5 _tvR pwd-LFcNFGXQ.	Fitness: 17
Generation: 3	String: 2 JNv#1Fs/2s@sA&euRn	Fitness: 16
Generation: 4	String: 2 JNv#1Fs/2s@sA&euRn	Fitness: 16
Generation: 5	String: U LdvR /u/-sG@A4eliv	Fitness: 15
Generation: 6	String: 5 LXvz Mw8-sf,4,eXQ.	Fitness: 14
Generation: 7	String: IXLn}n _e;ksC@iGeQ,(	Fitness: 13
Generation: 8	String: I Lnvk _efks@@iIeQ,v	Fitness: 12
Generation: 9	String: I Lnvk _efks@@iIeQ,v	Fitness: 12
Generation: 10	String: I Lnvk _efks@@iIeQ,v	Fitness: 12
Generation: 11	String: C Pnvn MeK sfoFGe&,F	Fitness: 11
Generation: 12	String: I ldv, Wz0?sfo Ge@k7	Fitness: 10
Generation: 13	String: I ldvn YeOksfoiGegQ(	Fitness: 9
Generation: 14	String: I A-vn Yh;ksfo;Gegks	Fitness: 8
Generation: 15	String: I A-vn Yh;ksfo;Gegks	Fitness: 8
Generation: 16	String: I A-vn Yh;ksfo;Gegks	Fitness: 8
Generation: 17	String: I L[v/ GeVksfmUGeQks	Fitness: 7
Generation: 18	String: I L[v/ GeVksfmUGeQks	Fitness: 7
Generation: 19	String: I L[v/ GeVksfmUGeQks	Fitness: 7
Generation: 20	String: I L[v/ GeVksfmUGeQks	Fitness: 7
Generation: 21	String: I 1-vK Ge;ksfoUGeLks	Fitness: 6
Generation: 22	String: I 1-vK Ge;ksfoUGeLks	Fitness: 6
Generation: 23	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 24	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 25	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 26	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 27	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 28	String: I gove G=5ksfo;GeQks	Fitness: 5
Generation: 29	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 30	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 31	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 32	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 33	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 34	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 35	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 36	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 37	String: I love GeOksfoUGeQks	Fitness: 4
Generation: 38	String: I love Ge2ksfo;GeCks	Fitness: 3
Generation: 39	String: I love Ge2ksfo;GeCks	Fitness: 3
Generation: 40	String: I love Ge2ksfo;GeCks	Fitness: 3
```

Sign:__________________