GOA COLLEGE OF ENGINEERING

"Bhausaheb Bandodkar Technical Education Complex"

FARMAGUDI- 403 401 GOA.

GOLDEN JUBILEE CELEBRATION

2016 - 2017

CERTIFICATE

Roll No. ____________ University Seat No. ____________

This is to Certify that Shri/Kum. ____ ____________________ of the ________________ Semester of four years Degree Course in ____ Engineering has completed the term work in the subject ________________________________within the four walls of **GOA COLLEGE OF ENGINEERING, FARMAGUDI** during the year.

Lecture In-charge	Head of the Dept.	Principal

CONTENTS

Sr. No.	Name of the Experiments	Date	Page	Signature
1	Introduction to Python	16-09-2020		
2	Introduction to Activation Functions and Implementation of an Artificial Neuron	23-09-2020		
3	Implementation of Logic Gates using Artificial Neurons	30-09-2020		
4	Design of an artificial Neuron using Hebbian Learning Rule	07-10-2020		
5	Design of an Artificial Neuron using Perceptron Learning Rule	14-10-2020		
6	Design of an Adaline	07-12-2020		
7	Design of an Artificial Neuron using Delta Learning Rule	09-12-2020		
8	Implementation of Roulette wheel Selection Technique	10-12-2020		
9	Clustering using Simple Competitive Learning Algorithm	14-12-2020		
10	Design of Brain-State-in-a-Box (BSB) Network	16-12-2020		

Exp no: 1 Date: 16-09-2020

Title: Introduction to python

Aim: To study the basics of python language

Programs used: Visual Studio Code, numpy, matplotlib

Theory: Python is an interpreted, high-level and general-purpose programming language. Python's design philosophy emphasizes code readability with its notable use of significant whitespace. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects.

NumPy is a general-purpose fundamental package for scientific computing with Python. It contains various features including these important ones:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Problem: Exercise 1:Data = 5649. Write a "while loop" to print data variable in reverse order

Exercise 2: Write a print statement to get following result.
name: <any name>   roll no: <roll no>   GATE rank: <rank>   percentile: <percentile>

Exercise 3: We have a list of players as follows.
players = ['abc, 'de', 'efg', 'ijk', 'lmn', 'op', 'qq', 'rr']
abc, lmn, qq, rr reached the semifinal of the tournament. Print the updated players list

Exercise 4: Write a simple program to print the first letter of your name using only asterisk '\' (as you might have done while learning loops in 'C' language programs)

Exercise 5:

I. Generate two arrays A1 and A2 of size 5 X 4 and 3 X 4 respectively using np.random()
II. Join them and make an array A3 of 8 X 4. Now append random numbers ranging between from 0 to 5 to make the fourth array A4 of size 10 X 10.
III. Print all the arrays and their transpose (Transpose of 'A' can be obtained by 'A.T')

Exercise 6: Create two dictionaries.
The first dictionary 'name' will contain first name(key) of a person and its hash value(value).
The Second will contain hash value(key) and mobile no(value).

1. Add 5 entries

2. Delete two entries by taking the input from user as the first name.
3. Add two entries by taking the input as the first name and mobile no.

Code:

```
'''
Introduction to python
'''
from pprint import pprint
from hashlib import md5
import numpy as np


#exe 1
DATA = 5649
ATAD = ''
while DATA > 0:
    ATAD += str(DATA%10)
    DATA = DATA//10
print(int(ATAD))

#exe 2
NAME = "Anirudha"
ROLL_NO = 171104008
GATE_RANK = 1
PERCENTILE = 99.99
print(f"name:{NAME} roll no:{ROLL_NO} GATE rank:{GATE_RANK} 
percentile:{PERCENTILE}")

#exe3
players = ['abc', 'de', 'ijk', 'efg', 'lmn', 'op', 'qq', 'rr']
for player in ['de', 'ijk', 'efg', 'op']:
    players.remove(player)
print(players)

#exe4
N = 6
for x in range(N):
    print(' ' * (N-x), end='')
    print('*', end='')
    if x ==3:
        print('******', end='')
    else:
        print(' ' * x * 2, end='')
    print('*')


#exe5
A1 = np.random.randn(5,4)
A2 = np.random.randn(3, 4)
A3 = np.concatenate((A1, A2))
A4 = np.concatenate((A3, np.random.randn(2, 4)))
A4 = np.concatenate((A4, np.random.randn(10, 6)), axis=1)
print(A4.shape)
# print(A1.T, A2.T, A3.T, A4.T)

#exe6
naava = ['Anirudha', 'Bnirudha', 'Cnirudha', 'Dnirudha', 
'Enirudha']
numbers = [1234567890, 2345678901, 3456789012, 4567890123, 
5678901234]
name = {naav:md5(naav.encode()).hexdigest() for naav in naava}
hashdict  = dict(zip(name.values(), numbers))
```

```python
for x in range(2):
    x = input('names to delete:')
    hashval = name[x]
    del name[x]
    del hashdict[hashval]
for x in range(2):
    x = input('name to add:')
    H = md5(x.encode()).hexdigest()
    name[x] = H
    num = input('mobile numbers:')
    hashdict[H] = num
pprint(name)
pprint(hashdict)
```

Conclusion: The basics of python programming language were studied

Result:

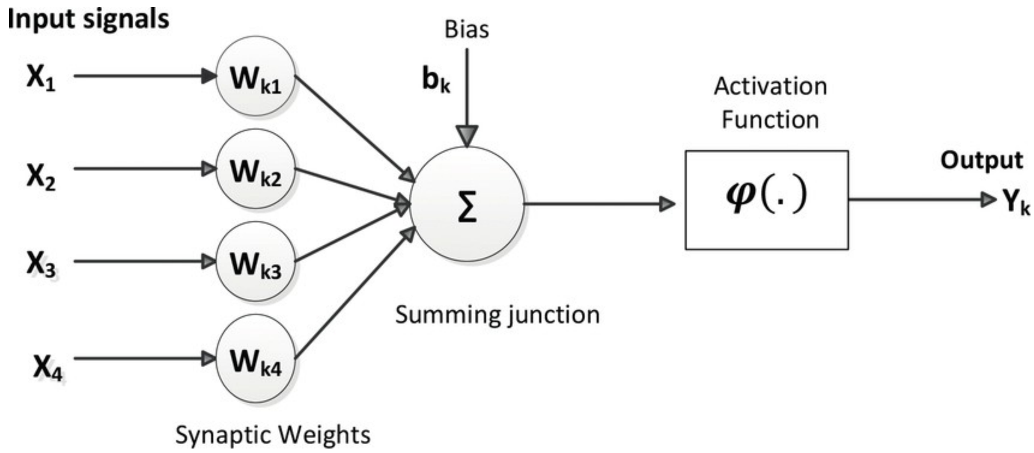
```
9465
name:Anirudha roll no:171104008 GATE rank:1 percentile:99.99
['abc', 'lmn', 'qq', 'rr']
      **
     *  *
    *    *
   ********
  *        *
 *          *
(10, 10)
names to delete:Anirudha
names to delete:Bnirudha
name to add:anirudha
mobile numbers:9999999999
name to add:bnirudha
mobile numbers:1234567890
{'Cnirudha': '29de093b14d53234b6b25e223dbbd803',
 'Dnirudha': '91827ace3a7f9b18cd713daf3ae30f89',
 'Enirudha': '01b61a2d4df5b005ed003fbba1751631',
 'anirudha': 'c8abbaaed5311a359b008c206b6690a1',
 'bnirudha': '4f6e34e1688014b10f10759cff4e1706'}
{'01b61a2d4df5b005ed003fbba1751631': 5678901234,
 '29de093b14d53234b6b25e223dbbd803': 3456789012,
 '4f6e34e1688014b10f10759cff4e1706': '1234567890',
 '91827ace3a7f9b18cd713daf3ae30f89': 4567890123,
 'c8abbaaed5311a359b008c206b6690a1': '9999999999'}
```

Exp no: 02 Date: 23-09-2020

Title: Introduction to activation functions and implementation of an Artificial Neuron

Aim: To observe the outputs of various activation functions with the help of implementation of an Artificial Neuron

Theory:



The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.

An artificial neuron is a mathematical function conceived as a model of biological neurons, a neural network. Artificial neurons are elementary units in an artificial neural network.
The artificial neuron receives one or more inputs (representing excitatory postsynaptic potentials and inhibitory postsynaptic potentials at neural dendrites) and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). Usually each input is separately weighted, and the sum is passed through a non-linear function known as an activation function or transfer function

It may be divided into 2 parts. The first part, g takes an input, performs an aggregation and based on the aggregated value the second part, f makes a decision.

Code:

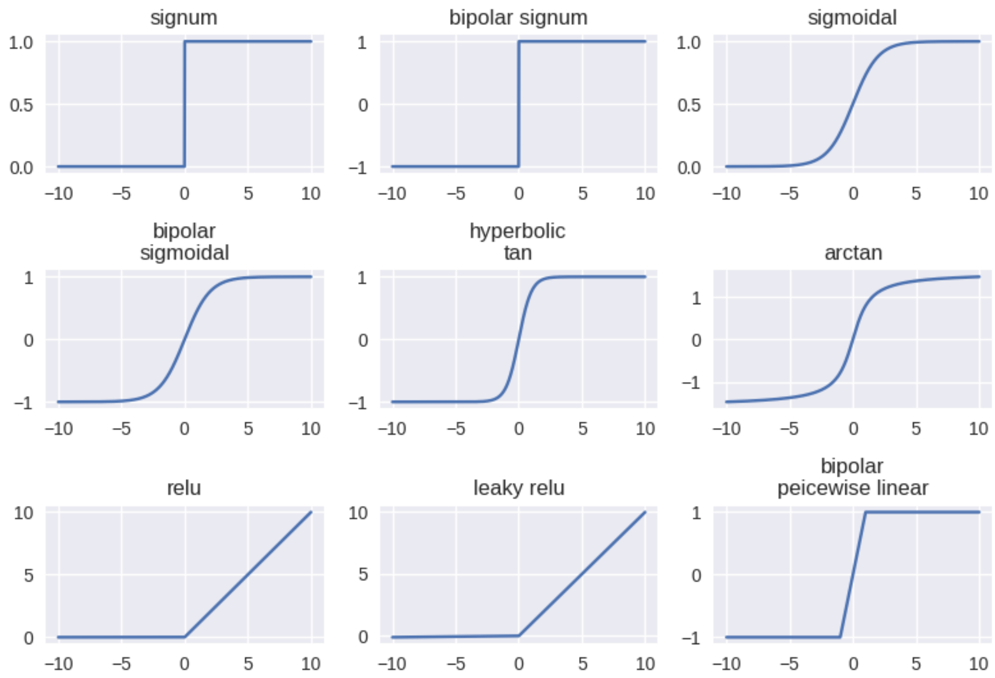
```
'''
implementation of neruron and activation functions
'''
import numpy as np


class Neuron:
    '''McCulloch pitts neuron model'''
    def __init__(self,activation_fn, weights, bias=0):
        self.weights = np.array(weights)
        self.inputs = np.empty_like(self.weights)
        self.act_fn = activation_fn
        self.bias = bias
```

```
    def calc_out(self):
        '''calculates neuron output using specified activation
function'''
        return self.act_fn(self.weights.T @ self.inputs +
self.bias)

def signum(net):
    '''signum activation function'''
    return net >= 0

def bipolar_step(net):
    '''bipolar step activation function'''
    return 2 * int(net > 0) -1

def u_peicewise_linear(net):
    '''unipolar peicewise linear activation function'''
    return min(1, max(0, net))

def b_peicewise_linear(net):
    '''bipolar peicewise linear activation function'''
    return min(1, max(-1, net))

def u_sigmoidal(net):
    '''unipolar sigmoidal activation function'''
    lam = 1
    return 1 / (1 + np.exp(-1 * lam * net))

def b_sigmoidal(net):
    '''bipolar sigmoidal activation function'''
    lam = 1
    return 2 / (1 + np.exp(-1 * lam * net)) - 1

def hyperbolic_tan(net):
    '''hyperolic tan activation function'''
    return np.tanh(net)

def arctan(net):
    '''arctan activation function'''
    return np.arctan(net)

def relu(net):
    '''relu activation function'''
    max(0, net)

def leaky_relu(net):
    '''leaky relu activation function'''
    max(0.01 * net, net)

#code for observing activation function outputs
import numpy as np
import matplotlib.pyplot as plt
from neuron import *


plt.style.use('Solarize_Light2')
fig, axs = plt.subplots(nrows=3, ncols=3)
ax = axs.flatten()
x = np.linspace(-10, 10, 1000)

y = signum(x)
ax[0].set_title('signum')
ax[0].plot(x, y)
```

```
y = bipolar_step(x)
ax[1].set_title('bipolar signum')
ax[1].plot(x, y)

y = u_sigmoidal(x)
ax[2].set_title('sigmoidal')
ax[2].plot(x, y)

y = b_sigmoidal(x)
ax[3].set_title('bipolar\nsigmoidal')
ax[3].plot(x, y)

y = hyperbolic_tan(x)
ax[4].set_title('hyperbolic\ntan')
ax[4].plot(x, y)

y = arctan(x)
ax[5].set_title('arctan')
ax[5].plot(x, y)

y = relu(x)
ax[6].set_title('relu')
ax[6].plot(x, y)

y = leaky_relu(x)
ax[7].set_title('leaky relu')
ax[7].plot(x, y)

y = b_peicewise_linear(x)
ax[8].set_title('bipolar\npeicewise linear')
ax[8].plot(x, y)

fig.tight_layout()
plt.show()
```

Conclusion: The working of various activation functions were studied and outputs observed

Result:



Exp no: 03 Date: 30-09-2020

Title: Implementation of Logic Gates using Artificial Neurons

Aim: To implement boolean logic gates using simple McCulloch-Pitts Model of Artificial Neuron

Code:

```
'''
logic gates
'''
import numpy
from neuron import Neuron, signum #Neuron class written in
practical 3

def show_output(gate, gate_inputs_list):
    '''
    helper function to display output
    '''
    for gate_input in gate_inputs_list:
        gate.inputs = numpy.array(gate_input)
        print(f'x1:{gate_input[0]} x2:{gate_input[1]} O:',
gate.calc_out())

gate_inputs = [[0,0],[0,1],[1,0], [1,1]] #inputs to test the
gates against

# and gate
print("\nand gate")
and_gate = Neuron(lambda x: x > 1, weights=[1, 1])
show_output(and_gate, gate_inputs)

#or_gate
print("\nor gate")
or_gate = Neuron(lambda x: x > 0, weights=[1, 1])
show_output(or_gate, gate_inputs)

# nand gate
print('\nnand')
nand_gate = Neuron(lambda x: x > -2, weights=[-1, -1])
show_output(nand_gate, gate_inputs)

# nor gate
print('\nnor')
nor_gate = Neuron(lambda x: x > -1, weights=[-1, -1])
show_output(nor_gate, gate_inputs)

# not
print('\nnot')
not_gate = Neuron(lambda x: x > -1, weights=[-1])
for g_inputs in [[0], [1]]:
    not_gate.inputs = numpy.array(g_inputs)
    print(f'x:{g_inputs} O:', not_gate.calc_out())


# xor
print('\nxor')
o1 = Neuron(signum,weights=[-2, 1],  bias=-1/2)
o2 = Neuron(signum,weights=[1, -1], bias=-1/2)
xor_gate = Neuron(signum, weights=[1, 1], bias=-1/2)

def sh_xor_output(gate_inputs_list):
    '''helper fuction to display xor output'''
```

```
    gate_inputs_array = numpy.array(gate_inputs_list)
    o1.inputs = o2.inputs = gate_inputs_array
    xor_gate.inputs = numpy.array([o1.calc_out(),
o2.calc_out()])
    print(f'x1:{gate_inputs_array[0]} x2:{gate_inputs_array[1]}
O:', xor_gate.calc_out())

for g_input in gate_inputs:
    sh_xor_output(g_input)
```

Conclusion: Logic gates were implemented using the McCulloch-Pitts Neuron model and outputs were observed

Result:

```
and gate
x1:0 x2:0 O: False
x1:0 x2:1 O: False
x1:1 x2:0 O: False
x1:1 x2:1 O: True

or gate
x1:0 x2:0 O: False
x1:0 x2:1 O: True
x1:1 x2:0 O: True
x1:1 x2:1 O: True

nand
x1:0 x2:0 O: True
x1:0 x2:1 O: True
x1:1 x2:0 O: True
x1:1 x2:1 O: False

nor
x1:0 x2:0 O: True
x1:0 x2:1 O: False
x1:1 x2:0 O: False
x1:1 x2:1 O: False

not
x:[0] O: True
x:[1] O: False

xor
x1:0 x2:0 O: False
x1:0 x2:1 O: True
x1:1 x2:0 O: True
x1:1 x2:1 O: False
```

Exp no: 04 Date: 07-09-2020

Title: Design of an Artificial Neuron using Hebbian Learning rule

Aim: To study and implement the Hebbian Learning rule

Theory: Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network. It is used for pattern classification. It is a single layer neural network, i.e. it has one input layer and one output layer. The input layer can have many units, say n. The output layer only has one unit. Hebbian rule works by updating the weights between neurons in the neural network for each training sample.
The weights change is calculated according to the following rule:

$$\Delta w = \eta . f(W^T X) . X$$

Problem:

Code:

```python
'''
hebbian learning rule
'''
import numpy
from neuron import Neuron, bipolar_step

weights = [1, -1, 0, 0.5]

inputs = [[1, -2, 1.5, 0], [1, -0.5, -2, -1.5], [0, 1, -1, 1.5]]
C = 1

neuron = Neuron(activation_fn=bipolar_step, weights=weights)
for epoch in range(5):
    print(f'\nepoch {epoch+1}')
    for x in inputs:
        neuron.inputs = numpy.array(x)
        o = neuron.calc_out()
        neuron.weights += C * o * neuron.inputs
        print(f'weights:\t{neuron.weights}')
```

Conclusion: The hebbian learning rule was studied and

Result:

```
epoch 1
weights:        [ 2.  -3.   1.5  0.5]
weights:        [ 1.  -2.5  3.5  2. ]
weights:        [ 1.  -3.5  4.5  0.5]

epoch 2
weights:        [ 2.  -5.5  6.   0.5]
weights:        [ 1. -5.  8.  2.]
weights:        [ 1.  -6.   9.   0.5]

epoch 3
weights:        [ 2.  -8.  10.5  0.5]
weights:        [ 1.  -7.5 12.5  2. ]
weights:        [ 1.  -8.5 13.5  0.5]

epoch 4
weights:        [  2.  -10.5  15.    0.5]
weights:        [  1. -10.  17.   2.]
weights:        [  1.  -11.   18.    0.5]

epoch 5
weights:        [  2.  -13.   19.5   0.5]
weights:        [  1.  -12.5  21.5   2. ]
weights:        [  1.  -13.5  22.5   0.5]
```

Exp no: 05 Date: 14-10-2020

Title: Design of an artificial Neuron using perceptron learning rule

Aim: To study and implement the perceptron learning rule

Theory: Perceptron is an algorithm for supervised learning of binary classifiers. A binary classifier is a function which can decide whether or not an input, represented by a vector of numbers, belongs to some specific class.
In perceptron learning rule weight change is calculated as:

$$\Delta w = \eta . (d_i - f(W^T X)) . X$$

Problem:

Code:

```
'''
perceptron learning rule
'''
import numpy
from neuron import Neuron, bipolar_step


np.set_printoptions(precision=2)
w = [1, -1, 0, 0.5]
C = 0.1
D = [-1, -1, 1]
X = numpy.array([[1, -2, 0, -1], [0, 1.5, -0.5, -1], [-1, 1,
0.5, -1]])

neuron = Neuron(activation_fn=bipolar_step, weights=w)

for epoch in range(5):
    print(f'\nepoch {epoch+1}')
    for x, di in zip(X, D):
        neuron.inputs = x
        o = neuron.calc_out()
        neuron.weights += C * (di- o) * x
        print(f'weights: {neuron.weights}')
```

Conclusion: Perceptron learning rule was implemented and oututs were observed

Result:

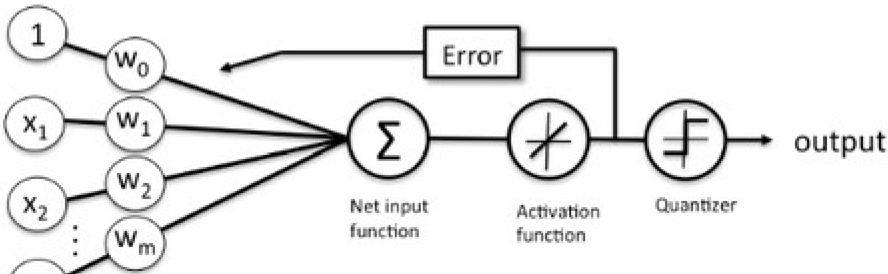
```
at soft_comp $ python pract_05.py

epoch 1
weights: [ 0.8 -0.6  0.   0.7]
weights: [ 0.8 -0.6  0.   0.7]
weights: [ 0.6 -0.4  0.1  0.5]

epoch 2
weights: [4.00e-01 5.55e-17 1.00e-01 7.00e-01]
weights: [4.00e-01 5.55e-17 1.00e-01 7.00e-01]
weights: [0.2 0.2 0.2 0.5]

epoch 3
weights: [0.2 0.2 0.2 0.5]
weights: [0.2 0.2 0.2 0.5]
weights: [5.55e-17 4.00e-01 3.00e-01 3.00e-01]

epoch 4
weights: [5.55e-17 4.00e-01 3.00e-01 3.00e-01]
weights: [5.55e-17 1.00e-01 4.00e-01 5.00e-01]
weights: [-0.2  0.3  0.5  0.3]

epoch 5
weights: [-0.2  0.3  0.5  0.3]
weights: [-0.2  0.3  0.5  0.3]
weights: [-0.2  0.3  0.5  0.3]
```

Exp no: 06 Date: 7-12-2020

Title: Design of an Adaline

Aim: To study and implement the Widrow-Hoff learning rule

Theory:



Adaline.

ADALINE (Adaptive Linear Neuron) is an early single-layer artificial neural network and the name of the physical device that implemented this network.
The difference between Adaline and the standard (McCulloch–Pitts) perceptron is that in the learning phase, the weights are adjusted according to the weighted sum of the inputs (the net).
In ADALINEs the weight change is calculated according to the widrow hoff learning rule given by:

$$\Delta W = \eta . (W^T X) . X$$

Problem:

Code:

```
'''
hebbian learning rule
'''
import numpy as np
from neuron import Neuron


w = [1, -1, 0, 0.5]
C = 0.1
D = [-1, -1, 1]
X = np.array([[1, -2, 0, -1], [0, 1.5, -0.5, -1], [-1, 1, 0.5, -
1]])

neuron = Neuron(activation_fn=lambda x: x, weights=w)
for epoch in range(5):
    print(f'\nepoch {epoch+1}')
    for x, di in zip(X, D):
        neuron.inputs = x
        o = neuron.calc_out()
        neuron.weights += C * o * x
        print(f'weights: {neuron.weights} f(net): {o} inputs:
{neuron.inputs}')
```

Conclusion: The widrow hoff learning rule was studied and ADALINE was implemented

Result:

```
epoch 1
weights: [ 1.25 -1.5   0.    0.25]
weights: [ 1.25 -1.88  0.12  0.5 ]
weights: [ 1.61 -2.23 -0.05  0.86]

epoch 2
weights: [ 2.13 -3.27 -0.05  0.33]
weights: [ 2.13 -4.06  0.21  0.86]
weights: [ 2.82 -4.75 -0.14  1.55]

epoch 3
weights: [ 3.9  -6.9  -0.14  0.47]
weights: [ 3.9  -8.52  0.4   1.55]
weights: [ 5.28 -9.9  -0.29  2.93]

epoch 4
weights: [  7.49 -14.32  -0.29   0.71]
weights: [  7.49 -17.63   0.81   2.92]
weights: [ 10.25 -20.39  -0.57   5.68]

epoch 5
weights: [ 14.79 -29.47  -0.57   1.14]
weights: [ 14.79 -36.23   1.68   5.65]
weights: [ 20.37 -41.81  -1.11  11.23]
```

Exp no: 07 Date: 09-12-2020

Title: Design of an artificial neuron using delta learning rule

Aim: To study and implement the delta learning rule

Theory: The delta learning rule is a gradient descent learning rule for updating the weights of the inputs to artificial neurons in a single-layer neural network. It is a special case of the more general backpropagation algorithm. For a neuron j with weight W_j and activation function $f(net)$, the change in weight is given by:

$$\Delta W = \eta . (d_i - f(net)) f'(net) X$$

$$where\ net = W^T X$$

Problem:

Code:

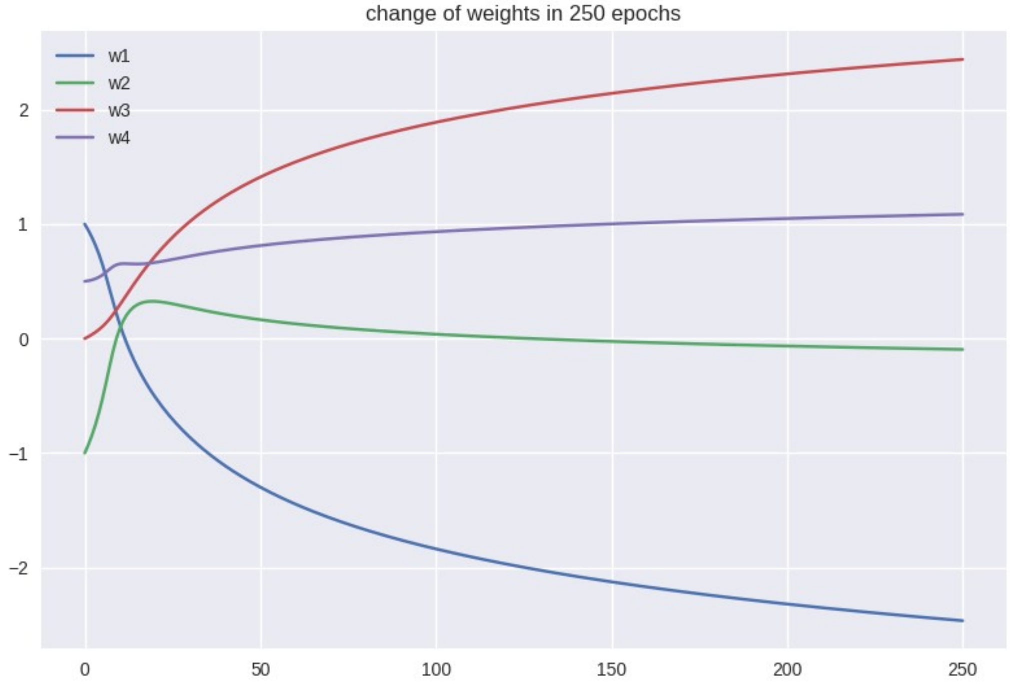
```
'''
Delta learning rule
'''
# from pprint import pprint
import numpy
import matplotlib.pyplot as plt
from neuron import Neuron, b_sigmoidal

numpy.set_printoptions(precision=2)

W = [1, -1, 0, 0.5]
C = 0.1
D = [-1, -1, 1]
X = numpy.array([[1, -2, 0, -1], [0, 1.5, -0.5, -1], [-1, 1,
0.5, -1]])
weight_change = [W]
neuron = Neuron(activation_fn=b_sigmoidal, weights=W)

for epoch in range(250):
    if epoch < 5:
        print(f'\nepoch: {epoch + 1}')
    for x, di in zip(X, D):
        neuron.inputs = x
        o = neuron.calc_out()
        neuron.weights += C * (di- o) * 0.5 * (1 - o*o) * x
        if epoch <5 :
            print(f'weights: {neuron.weights}\tf(net): {o:.2f}')
    weight_change.append(list(neuron.weights))

# pprint(weight_change)

plt.title('change of weights in 250 epochs')
plt.plot(weight_change)
plt.legend(['w1', 'w2', 'w3', 'w4'])
plt.show()
```

Conclusion: Delta learning rule was implemented and outputs were observed

Result:

```
epoch: 1
weights: [ 0.97 -0.95  0.    0.53]      f(net): 0.85
weights: [ 0.97 -0.96  0.    0.53]      f(net): -0.75
weights: [ 0.95 -0.93  0.02  0.5 ]      f(net): -0.84

epoch: 2
weights: [ 0.92 -0.87  0.02  0.53]      f(net): 0.82
weights: [ 0.92 -0.88  0.02  0.54]      f(net): -0.73
weights: [ 0.89 -0.85  0.03  0.51]      f(net): -0.82

epoch: 3
weights: [ 0.85 -0.78  0.03  0.55]      f(net): 0.78
weights: [ 0.85 -0.79  0.04  0.55]      f(net): -0.70
weights: [ 0.82 -0.76  0.05  0.52]      f(net): -0.80

epoch: 4
weights: [ 0.78 -0.67  0.05  0.56]      f(net): 0.72
weights: [ 0.78 -0.69  0.06  0.57]      f(net): -0.66
weights: [ 0.74 -0.65  0.08  0.54]      f(net): -0.76

epoch: 5
weights: [ 0.69 -0.56  0.08  0.58]      f(net): 0.64
```

```
weights: [ 0.69 -0.57  0.08  0.6 ]       f(net): -0.62
weights: [ 0.65 -0.53  0.1   0.55]       f(net): -0.72
```



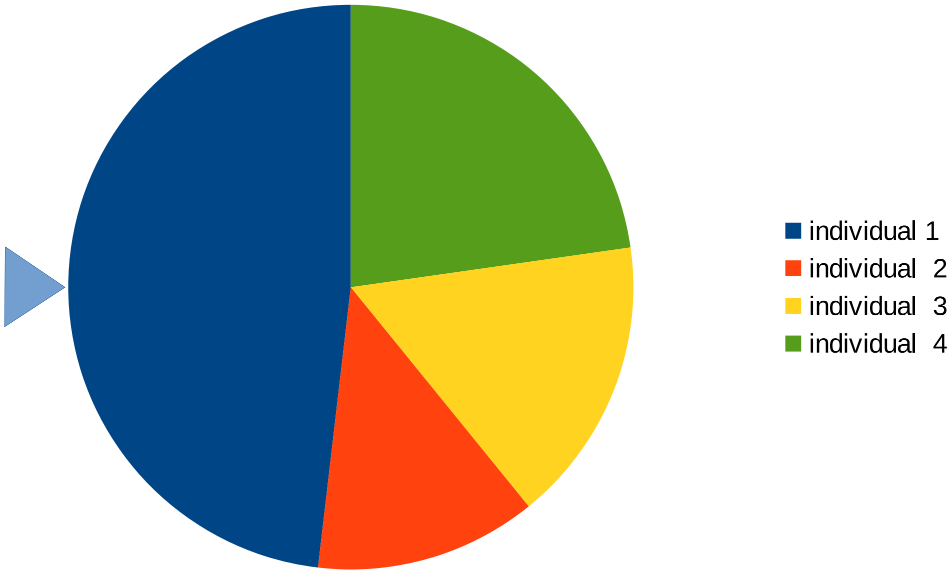
Exp no: 08 Date: 10-12-2020

Title: Implementation of roulette wheel selection

Aim: To study and implement roulette wheel selection technique

Theory: In this every individual can become a parent with a probability which is proportional to its fitness. Therefore, fitter individuals have a higher chance of mating and propagating their features to the next generation. Therefore, such a selection strategy applies a selection pressure to the more fit individuals in the population, evolving better individuals over time.

In a roulette wheel selection, the circular wheel is divided as described before. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated. The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.



Code:

```
'''
roulette wheel selection
'''

import numpy as np
import random
from pprint import pp

# initialise population
int1 = np.random.rand(10) * 10
int2 = np.random.rand(10) * 10
population = [(a, b) for a, b in zip(int1, int2)]

# goal of genetic algorithm is to maximise a*b
# fitness can be simply a*b
def fit(individual):
```

```
    a, b = individual
    return a * b

def rws(population):
    s = 0
    T = 0
    for individual in population:
        T += fit(individual)
    r = random.uniform(0, T)
    for individual in population:
        # visual representation of roulette wheel selection
        fitness_percent = int(fit(individual)/T * 100)
        print('█' * (fitness_percent-1)+'█', end='')
    print('\n'+' ' * int(r/T * 100)+'⮝')

    for p in population:
        s += fit(p)
        if s > r:
            return p

selected = rws(population)
print('phenotype:', selected)
print('fitness:', fit(selected))
```

Conclusion: Roulette wheel selection was studied and implemented

Result:

Exp no: 09 Date: 14-12-2020

Title: Clustering using simple competitive learning

Aim: To study and implement clustering using simple competitive learning algorithm

Theory: Simple competitive Network model accepts real valued vectors as inputs and consists of an
input layer with n nodes and output layer with n nodes. Every competitive node is described by a weight vector. A competition occurs among nodes in the outer layer, to find the winner node whose weight is
the closest to the input vector. Distance measure is usually Euclidean distance.

Problem:

Code:

```python
'''
simple competive learning
'''
import numpy as np


X = np.array([[1.1, 1.7, 1.8],
              [0.0, 0.0, 0.0],
              [0.0, 0.5, 1.5],
              [1.0, 0.0, 0.0],
              [0.5, 0.5, 0.5],
              [1.0, 1.0, 1.0]])

W = np.array([[0.2, 0.7, 0.3],
              [0.1, 0.1, 0.9],
              [1.0, 1.0, 1.0]])
n = 0.5
for epoch in range(5):
    print('\n###### epoch:', epoch+1, '######')
    for x in X:
        d = []
        for w in W:
            tmp = x-w
            d.append(tmp.T @ tmp)
        i = np.argmin(d)
        W[i] += 0.5 * (x - W[i])
    print(W, '\n')
```

Conclusion: Simple competitive learning algorithm was implemented and weight change was observed

Result:

```
###### epoch: 1 ######
[[0.525  0.3375 0.2875]
 [0.05   0.3    1.2   ]
 [1.025  1.175  1.2   ]]


###### epoch: 2 ######
[[0.565625  0.2921875 0.2859375]
 [0.025     0.4       1.35     ]
 [1.03125   1.21875   1.25     ]]


###### epoch: 3 ######
[[0.57070312 0.28652344 0.28574219]
 [0.0125     0.45       1.425     ]
 [1.0328125  1.2296875  1.2625    ]]


###### epoch: 4 ######
[[0.57133789 0.28581543 0.28571777]
 [0.00625    0.475      1.4625    ]
 [1.03320312 1.23242188 1.265625  ]]


###### epoch: 5 ######
[[0.57141724 0.28572693 0.28571472]
 [0.003125   0.4875     1.48125   ]
 [1.03330078 1.23310547 1.26640625]]
```

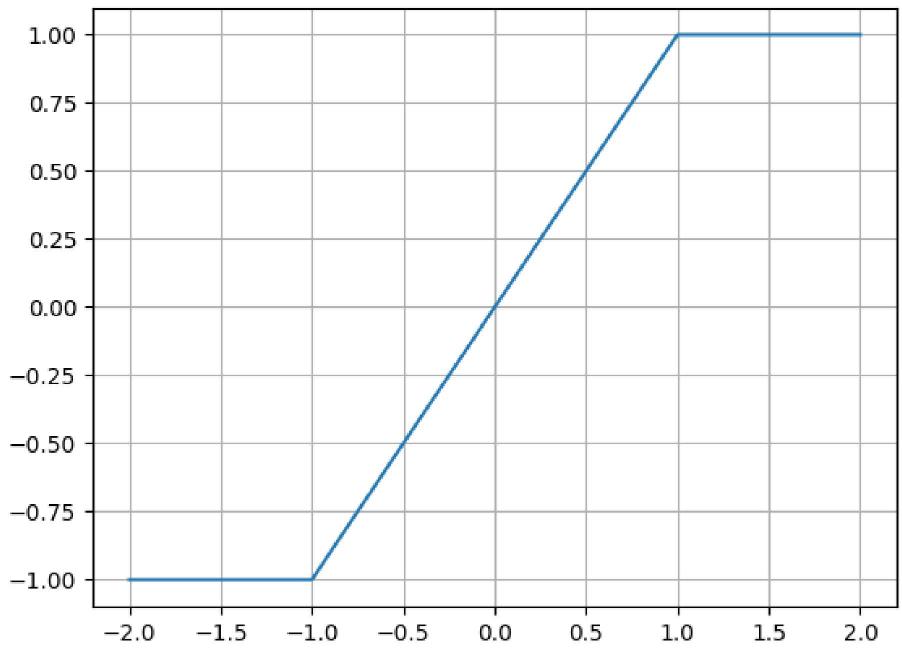
Exp no: 10 Date: 16-12-2020

Title: Design of a Brain-State-In-a-Box [BSB] Network

Aim: To study and implement a BSB network

Theory: A BSB network is fully connected, with as many nodes as the dimensionality n of the input space. All nodes are updated simultaneously, and the nodes take values in the continuous range from -1 to +1. The node function used is a ramp function

$$f(net) = min(1, max(-1, net))$$



which is bounded, continuous, and piecewise linear.
In the operation of this network, each node changes its state according to the following equation.

$$x_l(t+1) = f\left(\sum_{j=1}^{n} w_{l,j} x_j(t)\right)$$

where Xt (t) is the state of the .eth node at time t.
Each node' s activation belongs to the closed interval [- 1, 1], so that the state of the network always remains inside an n-dimensional “box”(hypercube), giving rise to the name of the network, “Brain-State-in-a-Box”.

Problem:

Code:

```python
'''
brain state in a box
'''
import numpy as np


training_set = np.array([[1, 1, 1], [-1, -1, -1], [1, -1, -1]])
P = len(training_set)
W = np.empty_like(training_set, dtype=float)
I = np.array([0.5, 0.6, 0.1])

def ramp(x):
    return np.maximum(-1, np.minimum(1, x))

for i in range(P):
    for j in range(P):
        W[i][j] = np.sum(training_set[:, i] * training_set[:,
j]) / P

print('weights:\n', W)
print('\ncorrupted input:\n', I)

while(I not in training_set):
    I = ramp(W.T @ I)

print('\ncorrected input:\n', I)
```

Conclusion: A BSB network was studied and implemented

Result:

```
weights:
 [[1.         0.33333333 0.33333333]
 [0.33333333 1.         1.        ]
 [0.33333333 1.         1.        ]]

corrupted input:
 [0.5 0.6 0.1]

corrected input:
 [1. 1. 1.]
```