|  |
| --- |
| **PRACTICAL FILE**  **BE (CSE) 7th Semester** |
| **NEURAL NETWORKS (CS 755B)**  **July 2023 – Dec 2023** |
| **Submitted By**  **Rohit Kumar Singh**  **Roll Number: UE203093** |
| **Submitted To**  **Dr. Sarbjeet Singh**  **Professor, Computer Science and Engineering** |
| **Computer Science and Engineering**  **University Institute of Engineering and Technology**  **Panjab University, Chandigarh – 160014, INDIA**  **2023** |

**INDEX**

| S. No. | Name of the Practical/Program | Page No. | Remarks |
| --- | --- | --- | --- |
| 1 | Implementation of McCulloch-Pitts Artificial Neuron Model and realization of AND function. | 3 - 5 |  |
| 2 | Realization of OR, NOT, and NOR functions using McCulloch-Pitts Neuron Model and display of decision boundary. | 6 - 11 |  |
| 3 | Build a network of McCulloch–Pitts units capable of computing the parity function of two, three, and four given bits. | 12 - 13 |  |
| 4 | Design a McCulloch–Pitts unit capable of recognizing the letter “T” digitized in a 10 × 10 array of pixels. Dark pixels should be coded as ones and white pixels as zeroes. | 14 - 17 |  |
| 5 | Build a recurrent network capable of adding two sequential streams of bits of arbitrary finite length. Appropriately assume any required information yourself. | 18-20 |  |
| 6 | Convert the units to feet and inches for height, and kilograms for weight. Perform regression analysis using the normal equation method by taking into account that size of data which works best as per your system's configuration. | 21-23 |  |
| 7 | Perform regression analysis using the Gradient Descent approach on height and weight data. | 24-27 |  |
| 8 | Realization of AND and OR functions using Hebbian Learning and display of decision boundary. | 28-30 |  |
| 9 | Write a program to demonstrate competitive learning and display the results in a geometric format. | 31-33 |  |
| 10 | Write a program to demonstrate Boltzmann Learning. | 34-35 |  |
| 11 | Write a program to solve XOR problem using Multi-Layer Perceptron and draw the decision boundary. | 36-37 |  |
| 12 | Write a Python program to implementation the Backpropagation Algorithm. | 38-39 |  |
|  |  |  |  |
|  |  |  |  |

**Practical 1:**

**Implementation of McCulloch-Pitts Artificial Neuron Model and realization of AND function.**

**THEORETICAL DISCUSSION:**

The main elements of the McCulloch-Pitts model can be summarized as follow:

1. Neuron activation is binary. A neuron either fire or not-fire.
2. For a neuron to fire, the weighted sum of inputs has to be equal or larger than a predefined threshold.
3. If one or more inputs are inhibitory the neuron will not fire.
4. It takes a fixed one-time step for the signal to pass through a link.
5. Neither the structure nor the weights change over time.

McCulloch and Pitts developed a mathematical formulation known as linear threshold gate, which describes the activity of a single neuron with two states, firing and not-firing. In its simplest form, the mathematical formulation is as follows:

Sum =

Y(Sum) = 1, if Sum >= T

= 0, otherwise

Where I1, I2, …, In are binary input values ∈ {0, 1}; W1, W2, …, Wn are weights associated with each input ∈ {−1,1}; Sum is the weighted sum of inputs; and T is a predefined threshold value for the neuron activation (i.e., firing).

* The AND function is “activated” when both the incoming inputs is “on”. In “neural” terms, the neuron fires when both the incoming signals are excitatory.

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

Since we want the neuron to fire only when both inputs are excitatory, the threshold for activation must be 2. To obtain an output of 2, we need both inputs to be excitatory, therefore, the weights must be positive (i.e., 1).

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Importing NumPy**

import numpy as np

**# 2. Defining utility functions**

*# Function for calculating weighted sum*

def **calc\_weighted\_sum**(x, w) :

    weighted\_sum = []

    for vector in x :

**sum** = 0

        for i in **range**(vector.size) :

**sum** = **sum** + (vector[i]\*w[i])

        weighted\_sum.append(**sum**)

    ws = np.array(weighted\_sum)

    return ws

*# Function for checking activation of neuron*

def **activation**(t, input\_vector) :

    if input\_vector >= t :

        return 1

    else :

        return 0

**# 3. Generating Input and weight vectors**

*# Generating input vector*

x1 = np.random.randint(0, 2, size=(5, 4))

*# Generating weights*

w1 = np.random.randint(-1, 2, size=(4))

**print**(f'Input vector : \n {x1}\n')

**print**(f'Weights : \n {w1}')

*# Setting threshold*

t = 0

*# Calculating weighted sum*

ws1 = calc\_weighted\_sum(x1, w1)

*# Checking activation for each input*

for i in ws1 :

**print**("Activation : " + str(activation(t, i)))

**# 4. Realization of AND Function**

*# Input vector*

x2 = np.array([[0, 0],

              [0, 1],

              [1, 0],

              [1, 1]])

*# Setting weights*

w2 = np.array([1, 1])

**print**(f'Input vector : \n {x2}\n')

**print**(f'Weights : \n {w2}')

*# Setting threshold*

t = 2

*# Calculating weighted sum*

ws2 = calc\_weighted\_sum(x2, w2)

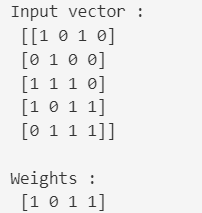
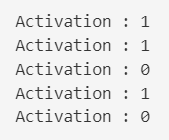
*# Checking activation for each input*

for i in ws2 :

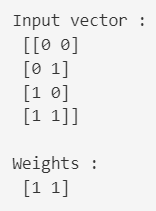
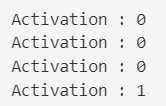
**print**("Activation : " + str(activation(t, i)))

**OUTPUT:**

For general binary input vectors:

** **

Realization of AND function:

** **

**DISCUSSION ON OUTPUT:**

As expected, only the last input, with 1 as both input values and with weight being 1 for each value resulted in the dot product >= set threshold. Thus, the neuron fired.

**Practical 2:**

**Realization of OR, NOT and NOR functions using McCulloch-Pitts Neuron Model and display of decision boundary.**

**THEORETICAL DISCUSSION:**

We will use the McCulloch-Pitts model to replicate the behavior of the required boolean functions, as expressed in their respective truth tables.

* The OR function is “activated” when at least one of the incoming inputs is “on”. In “neural” terms, the neuron fires when at least one of the incoming signals is excitatory.

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Since we want the neuron to fire when at least one of the inputs is excitatory, the threshold for activation must be 1. To obtain an output of at least 1, we need both inputs to be excitatory, therefore, the weights must be positive (i.e., 1).

* The NOT function is “activated” when the incoming input is “off”. In “neural” terms, the neuron fires when the single input signal is inhibitory.

|  |  |
| --- | --- |
| **x** | **y** |
| 0 | 1 |
| 1 | 0 |

Since we want the neuron to fire only when the input is inhibitory, the threshold for activation must be 0. To obtain an output of 0, we need the input to be inhibitory, therefore, the weight must be negative (i.e., -1).

* The NOR function is “activated” when all the incoming inputs are “off”. In this sense, it is the inverse of the OR function. In “neural” terms, the neuron fires when all the signals are inhibitory.

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

Since we want the neuron to fire only when both inputs are inhibitory, the threshold for activation must be 0. To obtain an output of 0, we need both inputs to be inhibitory, therefore, the weights must be negative (i.e., -1).

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Importing Libraries**

import numpy as np

import matplotlib.pyplot as plt

**# 2. Defining utility functions**

*# Function for calculating weighted sum*

def **calc\_weighted\_sum**(x, w) :

    weighted\_sum = []

    for vector in x :

**sum** = 0

        for i in **range**(vector.size) :

**sum** = **sum** + (vector[i]\*w[i])

        weighted\_sum.append(**sum**)

    ws = np.array(weighted\_sum)

    return ws

*# Function for checking activation of neuron*

def **activation**(t, ws) :

    if ws >= t :

        return 1

    else :

        return 0

*# Function for drawing decision boundary*

def **draw\_boundary**(t, x, w) :

    if (t == 0) :

        try :

*# Plotting all possible binary inputs*

            for i, j in x :

                plt.plot(i, j, 'ro')

*# Plotting decision boundary*

            plt.axline((-1, 1), (1, -1))

        except :

*# Plotting all possible binary inputs*

            for i in x :

                plt.plot(i, 0, 'ro')

*# Plotting decision boundary*

            plt.axline((0, 10), (0, -10))

    else :

        try :

*# Plotting all possible binary inputs*

            for i, j in x :

                plt.plot(i, j, 'ro')

*# Plotting decision boundary*

            plt.axline((0, t), (t, 0))

        except :

*# Plotting all possible binary inputs*

            for i in x :

                plt.plot(i, 0, 'ro')

*# Plotting decision boundary*

            plt.axline((t, -10), (t, 10))

*# Setting coefficients of x and y for inequality*

    try :

        a = w[0]

        b = w[1]

        c = t

    except :

        a = w

        b = 0

        c = t

*# Creating a mesh of x and y values*

    x = np.linspace(-1, 10, 1000)

    y = np.linspace(-1, 10, 1000)

    X, Y = np.meshgrid(x, y)

*# Defining the inequality as a boolean mask*

    inequality = (a\*X + b\*Y >= c)

*# Displaying the shaded region for the inequality*

    plt.imshow(inequality, extent=(x.min(), x.max(), y.min(), y.max()), origin="lower", cmap="Blues", alpha=0.7)

    plt.xlabel('x')

    plt.ylabel('y')

    plt.grid(True)

    plt.show()

**# 3. Realization of OR Function**

*# Input vector*

x1 = np.array([[0, 0],

              [0, 1],

              [1, 0],

              [1, 1]])

*# Setting weights*

w1 = np.array([1, 1])

**print**(f'Input vector : \n {x1}\n')

**print**(f'Weights : \n {w1}')

*# Setting threshold*

t = 1

*# Calculating weighted sum*

ws1 = calc\_weighted\_sum(x1, w1)

*# Checking activation for each input*

for i in ws1 :

**print**("Activation : " + str(activation(t, i)))

*# Drawing decision boundary*

draw\_boundary(t, x1, w1)

**# 4. Realization of NOT Function**

*# Input vector*

x2 = np.array([[0],

              [1]])

*# Setting weights*

w2 = np.array([-1])

**print**(f'Input vector : \n {x2}\n')

**print**(f'Weights : \n {w2}')

*# Setting threshold*

t = 0

*# Calculating weighted sum*

ws2 = calc\_weighted\_sum(x2, w2)

*# Checking activation for each input*

for i in ws2 :

**print**("Activation : " + str(activation(t, i)))

*# Drawing decision boundary*

draw\_boundary(t, x2, w2)

**# 5. Realization of NOR Function**

*# Input vector*

x3 = np.array([[0, 0],

              [0, 1],

              [1, 0],

              [1, 1]])

*# Setting weights*

w3 = np.array([-1, -1])

**print**(f'Input vector : \n {x3}\n')

**print**(f'Weights : \n {w3}')

*# Setting threshold*

t = 0

*# Calculating weighted sum*

ws3 = calc\_weighted\_sum(x3, w3)

*# Checking activation for each input*

for i in ws3 :

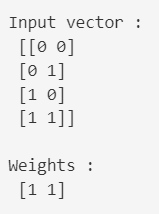
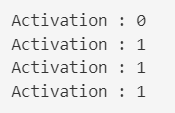
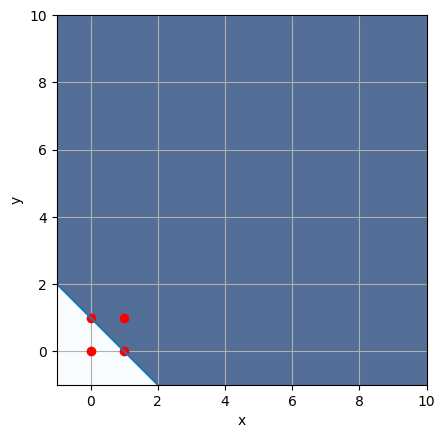
**print**("Activation : " + str(activation(t, i)))

*# Drawing decision boundary*

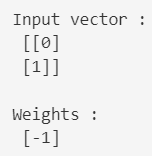
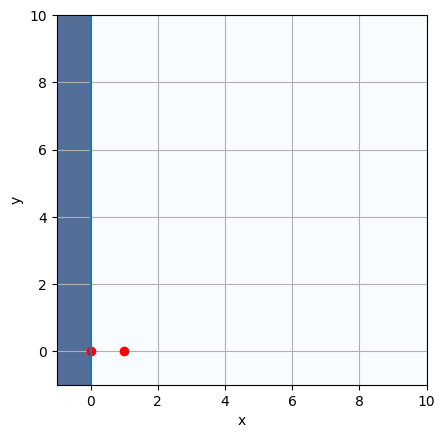
draw\_boundary(t, x3, w3)

**OUTPUT:**

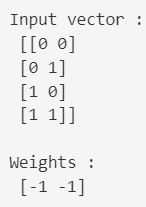
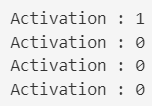
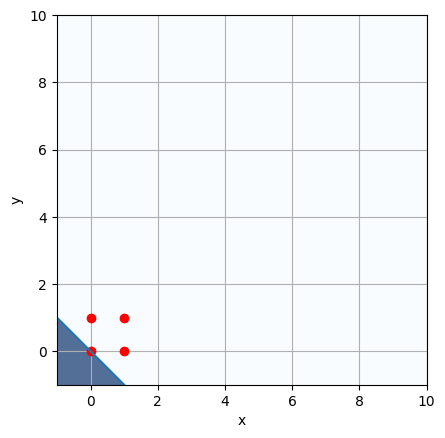
Realization of OR function:

****  ** **

Realization of NOT function:

Realization of NOR function:

**  **

**DISCUSSION ON OUTPUT:**

In the case of OR function, with at least one input as 1 and with weights being 1 for each value, the neuron is fired.

In the case of NOT function, with input as 0 and with weight being -1, the neuron is fired.

In the case of NOR function, with both the input as 0 and with weights being -1 for each value, the neuron is fired.

**Practical 3:**

**Build a network of McCulloch–Pitts units capable of computing the parity function of two, three, and four given bits.**

**THEORETICAL DISCUSSION:**

The parity of n given bits is 1 if an odd number of them is equal to 1, otherwise it is 0.

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Importing NumPy**

import numpy as np

**# 2. Defining utility functions**

*# Function for calculating sum*

def calc\_sum(x, w) :

    sum = 0

    for i in range(x.size) :

        sum = sum + (x[i]\*w)

    return sum

# Function for checking parity

def parity(ws) :

    if (ws % 2 == 1) :

        return 1

    else :

        return 0

**# 3. Generating random binary input vectors of size 2, 3 and 4**

*# Input vector*

x1 = np.random.randint(0, 2, size=(2, 1))

x2 = np.random.randint(0, 2, size=(3, 1))

x3 = np.random.randint(0, 2, size=(4, 1))

print(x1, "\n")

print(x2, "\n")

print(x3, "\n")

# Setting weights

w = np.array([3])

**# 4. Checking parity for each input vector**

*# Calculating sum for each vector and storing in list*

ws = []

ws1 = calc\_sum(x1, w)

ws.append(ws1)

ws2 = calc\_sum(x2, w)

ws.append(ws2)

ws3 = calc\_sum(x3, w)

ws.append(ws3)

ws = np.array(ws)

print(ws)

# Checking if Odd parity or not

for s in ws :

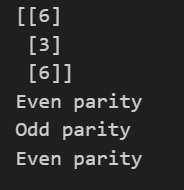
    if (parity(s) == 1) :

        print("Odd parity")

    else :

        print("Even parity")

**OUTPUT:**

****

**DISCUSSION ON OUTPUT:**

As expected, the vector containing odd number of 1s has odd parity and even parity otherwise.

**Practical 4:**

**Design a McCulloch–Pitts unit capable of recognizing the letter “T” digitized in a 10 × 10 array of pixels. Dark pixels should be coded as ones and white pixels as zeroes.**

**THEORETICAL DISCUSSION:**

To recognize the letter ‘T’ we can take a 10**\***10 weight matrix which contains 1 at the dark pixels and 0 at the white pixels. The first two rows and the middle two columns are coded as 1 and the rest with 0.

We then take an input matrix of 10\*10 size with entries as 0 or 1. We see that we can multiply the corresponding bits of the weight matrix and the input matrix and add them to get a weighted sum.

If the input matrix corresponds to a perfect ‘T’ then the sum will be maximum and equal to 36. Otherwise, the sum will be less than this. We can set a threshold value like 30 so that if the weighted sum of the input matrix is greater than or equal to 30, it is recognized as the letter ‘T’ and otherwise not.

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Importing Libraries**

import matplotlib.pyplot as plt

import matplotlib.colors

**# 2. Defining utility functions**

# Function for calculating weighted sum

def calc\_weighted\_sum(T, input) :

    sum = 0

    for i in range(len(input)) :

        for j in range(len(input[0])) :

            sum += input[i][j]\*T[i][j]

    return sum

# Function for checking activation of neuron

def recognition(sum, t) :

    if sum >= t :

        return 1

    else :

        return 0

**# 3. Defining 10\*10 array for the letter 'T'**

T = [

    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0]

]

cmap = matplotlib.colors.ListedColormap(["white", "black"])

plt.subplot(121).imshow(T, cmap = cmap)

plt.title("Letter T")

**# 4. Recognising the letter 'T' in the input 10\*10 array**

# Higher the threshold, more accurate is the letter

# For given configuration of letter 'T', max threshold can be set to 36

threshold = 30

# Input 1 that closely resembles the letter 'T'

input1 = [

    [1, 0, 1, 1, 1, 1, 0, 1, 0, 1],

    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0]

]

# Check for Input 1

sum1 = calc\_weighted\_sum(T, input1)

if(recognition(sum1, threshold)) :

    cmap = matplotlib.colors.ListedColormap(["white", "black"])

    plt.subplot(121).imshow(input1, cmap = cmap)

    plt.title("Input 1 : T recognised")

else :

    cmap = matplotlib.colors.ListedColormap(["white", "red"])

    plt.subplot(121).imshow(input1, cmap = cmap)

    plt.title("Input 1 : T unrecognised")

# Input 2 that doesn't closely resemble the letter 'T'

input2 = [

    [1, 0, 1, 1, 0, 1, 0, 1, 0, 1],

    [1, 1, 1, 1, 1, 1, 0, 1, 1, 1],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 0, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0],

    [0, 0, 0, 0, 1, 1, 0, 0, 0, 0]

]

# Check for Input 2

sum2 = calc\_weighted\_sum(T, input2)

if(recognition(sum2, threshold)) :

    cmap = matplotlib.colors.ListedColormap(["white", "black"])

    plt.subplot(122).imshow(input2, cmap = cmap)

    plt.title("Input 2 : T recognised")

else :

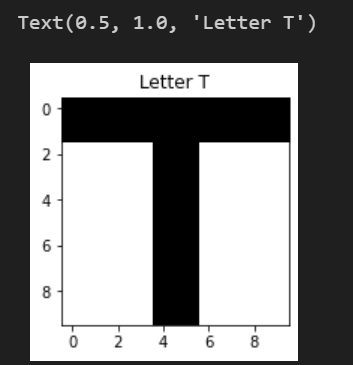
    cmap = matplotlib.colors.ListedColormap(["white", "red"])

    plt.subplot(122).imshow(input2, cmap = cmap)

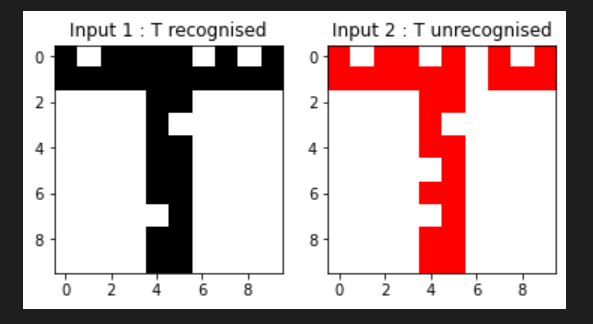
    plt.title("Input 2 : T unrecognised")

**OUTPUT:**

The actual digitized letter ‘T’.

****

Recognition of various inputs.



**DISCUSSION ON OUTPUT:**

The weighted sum for the first input turns out to be 31 which exceeds the threshold of 30, hence it is recognized as ‘T’.

The weighted sum for the first input turns out to be 28 which does not exceed the threshold of 30, hence it is not recognised as ‘T’.

**Practical 5:**

**Build a recurrent network capable of adding two sequential streams of bits of arbitrary finite length. Appropriately assume any required information yourself.**

**Theoretical Discussion:**

The approach involves defining weights for bit addition and simulating the carry mechanism. The code iterates through the bit streams, employing weighted sums to calculate additions and carries. It includes defining a function for bit addition, handling carry propagation, and utilizing weighted sums for binary addition. It processes bit streams in reverse order, ensuring proper carry propagation. Lastly, the theoretical approach highlights the significance of initializing and maintaining carry values between iterations to ensure the accurate addition of sequential bit streams.

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**#1. Importing Libraries**

import numpy as np

from itertools import zip\_longest

**#2. Defining utility functions**

# Function for adding bits using the weighted sum

def bit\_adder(ws):

    value = 0

    if ws % 2 == 1:

        value = 1

    carry = 0

    if ws >= 2:

        carry = 1

    print(f'\t\tValue : {value}  Carry : {carry}')

    return value, carry

#3. Taking input two sequential streams of bits of arbitrary finite length

# Taking 1st input stream of arbitrary finite length

stream1 = np.array([int(x) for x in input("Enter 1st stream : ").split()])

print(f'Stream1 : {stream1}')

# Taking 2nd input stream of arbitrary finite length

stream2 = np.array([int(x) for x in input("Enter 2nd stream : ").split()])

print(f'Stream2 : {stream2}')

A black background with white text

Description automatically generated

#4 Defining weights, carry and result variables

# Defining weights for adding bits

weights = np.array([1, 1, 1])

# Variable to store the carry generated by current addition operation and pass it to next addition operation

carry = 0

# Defining result stream

res = []

#5. Adding the streams bitwise

# Reversing the bit-streams as we add the bits starting from right towards left

rev\_stream1 = reversed(stream1)

rev\_stream2 = reversed(stream2)

# Making both bit-streams of equal length by adding redundant zeroes

streams = zip\_longest(rev\_stream1, rev\_stream2, fillvalue = 0)

# Adding the bits of streams

for bits in streams:

    # Displaying the bits from streams along with carry-bit for each iteration

    print([bits[0], bits[1], carry])

    # Calculating the weighted sum of bits and carry

    ws = [bits[0], bits[1], carry] @ weights

    # Calculating the value after addition and next carry

    value, carry = bit\_adder(ws)

    # Appending value to the result array

    res.append(value)

# Adding the last carry to result if any

if carry == 1:

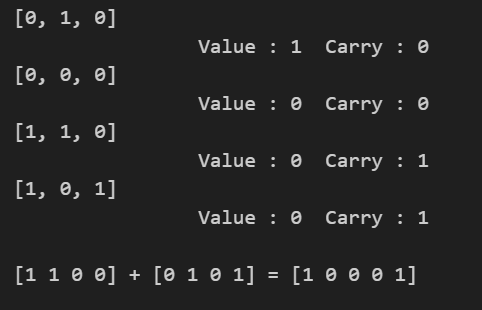
    res.append(carry)

# Reversing the array to get added stream in correct order

res = np.array(list(reversed(res)))

print(f"\n{stream1} + {stream2} = {res}")

**Output:**



**DISCUSSION ON OUTPUT:**

The output showcases the binary addition of two input streams, revealing the accumulated results of sequential bit-wise addition along with carry propagation, demonstrating a basic neural network approach for arithmetic operations.

**PRACTICAL 6:**

**Convert the units to feet and inches for height, and kilograms for weight. Perform regression analysis using normal equation method by taking into account that size of data which works best as per your system's configuration.**

**THEORETICAL DISCUSSION:** Regression is a supervised machine-learning technique that is used to predict continuous values. The ultimate goal of the regression algorithm is to plot a best-fit line or a curve between the data. The three main metrics that are used for evaluating the trained regression model are variance, bias, and error.

**IMPLEMENTATION DETAILS/SOURCE CODE:**

import pandas as pd

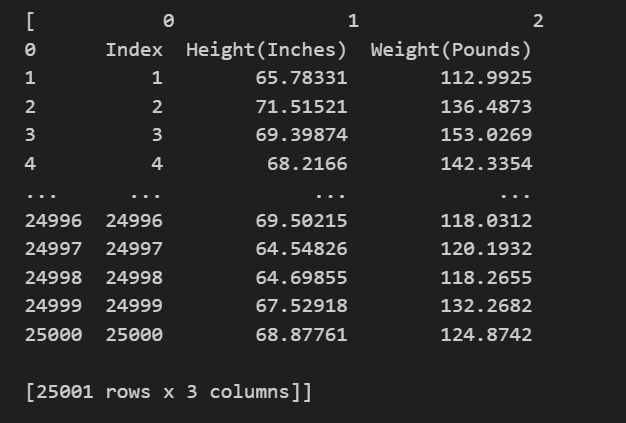
import numpy as np

# Load data from CSV

data = pd.read\_html(

    'http://socr.ucla.edu/docs/resources/SOCR\_Data/SOCR\_Data\_Dinov\_020108\_HeightsWeights.html')

print(data)



* **Converting height from one unit to another unit**

c = 1

for i in df[1]:

  if i != "Height(Inches)":

    # print(i)

    k = i

    k = float(k)

    k = k/12

    # print(k)

    df[1][c] = k

    c = c+1

  else:

    df[1][0] = "Height(Feet)"

* Converting weight from one unit to another unit

c = 1

for i in df[2]:

  if i != "Weight(Pounds)":

    # print(i)

    k = i

    k = float(k)

    k = k\*0.45359237

    # print(k)

    df[2][c] = k

    c = c+1

  else:

    df[2][0] = "Weight(Kg)"

heights = []

for i in df[1]:

  if i != "Height(Feet)":

    # print(i)

    k = i

    k = float(k)

    # print(k)

    heights.append(k)

weights = []

for i in df[2]:

  if i != "Weight(Kg)":

    k = i

    k = float(k)

    weights.append(k)

mean\_height = sum(heights) / len(heights)

mean\_weight = sum(weights) / len(weights)

# Calculate the coefficients

numerator = sum((heights[i] - mean\_height) \* (weights[i] - mean\_weight) for i in range(len(heights)))

denominator = sum((heights[i] - mean\_height) \*\* 2 for i in range(len(heights)))

theta\_1 = numerator / denominator

theta\_0 = mean\_weight - theta\_1 \* mean\_height

print("Optimal θ0:", theta\_0)

print("Optimal θ1:", theta\_1)

# Calculate predicted weights

predicted\_weights = [theta\_0 + theta\_1 \* h for h in heights]

# Plot the data points and the regression line

plt.scatter(heights, weights, label="Original Data")

plt.plot(heights, predicted\_weights, color='red', label="Regression Line")

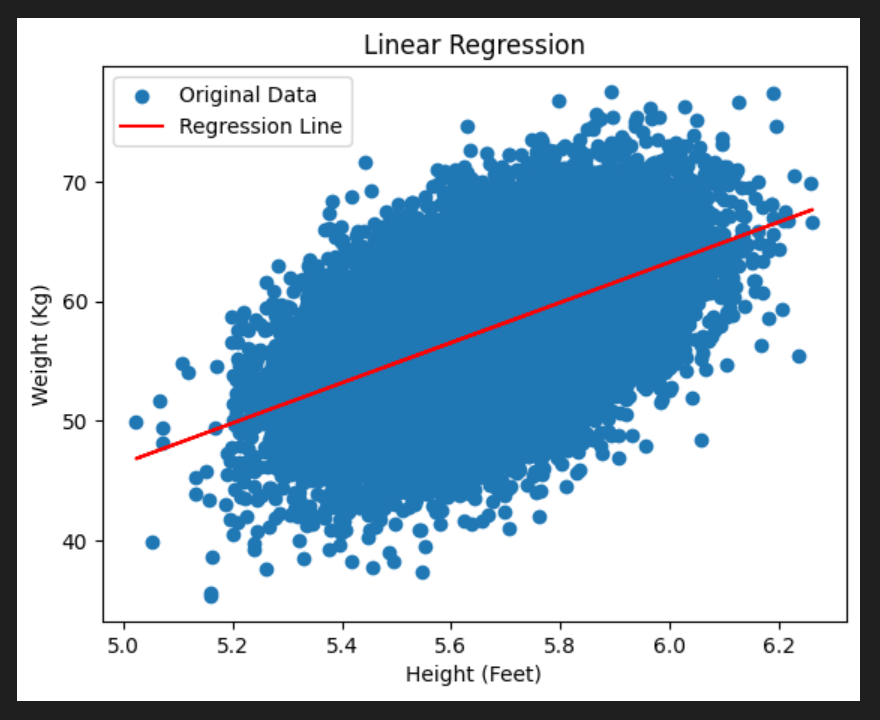
plt.xlabel("Height (Feet)")

plt.ylabel("Weight (Kg)")

plt.title("Linear Regression")

plt.legend()

plt.show()



**Practical 7:**

**Perform regression analysis using Gradient Descent approach on height and weight data.**

**THEORETICAL DISCUSSION:**

Linear regression using gradient descent is an optimization technique for finding the coefficients (c and m) that minimize the cost function, which measures the error between predicted and observed values in a linear regression model.

In gradient descent:

1. Initialize c and m with random values.

2. Calculate the predicted values (Ypred) using the current c and m.

3. Compute the gradient of the cost function with respect to c and m.

4. Update c and m by moving them in the opposite direction of the gradient to minimize the cost.

5. Repeat steps 2-4 iteratively until convergence or a predefined number of iterations.

The cost function for linear regression is typically the Mean Squared Error (MSE), which quantifies the average squared difference between observed and predicted values:

- n is the number of data points.

- Yi is the observed value.

- Xi is the corresponding independent variable value.

The gradient descent updates for c and m are performed using the partial derivatives of the MSE:

Where:

- α (alpha) is the learning rate, a hyperparameter that controls the step size in each iteration.

Repeat these updates until the cost converges to a minimum value, indicating that the coefficients c and m have been optimized. Once converged, the optimized c and m values can be used to make predictions for new data points using the linear regression model:

Ypred = c + m \* Xnew

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Import libraries**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

**# 2. Loading data**

data = pd.read\_html('http://socr.ucla.edu/docs/resources/SOCR\_Data/SOCR\_Data\_Dinov\_020108\_HeightsWeights.html')

**# 3. Preprocessing the data**

# Select the first table from the loaded data

df = data[0]

# Convert height from inches to feet

df[1] = df[1].apply(lambda x: float(x) / 12 if x != "Height(Inches)" else "Height(Feet)")

# Convert weight from pounds to kilograms

df[2] = df[2].apply(lambda x: float(x) \* 0.45359237 if x != "Weight(Pounds)" else "Weight(Kg)")

# Extract heights and weights as lists

heights = [float(h) for h in df[1] if h != "Height(Feet)"]

weights = [float(w) for w in df[2] if w != "Weight(Kg)"]

**# 4. Calculation of linear regression using gradient descent approach**

# Convert lists to numpy arrays

X = np.array(heights).reshape(-1, 1)

y = np.array(weights)

# Hyperparameters

learning\_rate = 0.000001

num\_iterations = 1000

batch\_size = 100

# Initialize coefficients with approx values

theta = np.array([-40.0, 15.0])

# Optimization using Gradient Descent

for \_ in range(num\_iterations):

    for batch\_start in range(0, len(X), batch\_size):

        # Create a mini-batch

        X\_batch = X[batch\_start:batch\_start+batch\_size]

        y\_batch = y[batch\_start:batch\_start+batch\_size]

        # Calculate predictions

        predictions = theta[0] + theta[1] \* X\_batch

        # Calculate the error

        error = predictions - y\_batch

        # Update coefficients using gradient descent

        gradient\_0 = (1 / len(X\_batch)) \* np.sum(error)

        gradient\_1 = (1 / len(X\_batch)) \* np.sum(error \* X\_batch)

        theta[0] -= (learning\_rate \* gradient\_0)

        theta[1] -= (learning\_rate \* gradient\_1)

# Extract coefficients

intercept = theta[0]

slope = theta[1]

# Print the calculated optimized coefficients

print(f'Optimal (Intercept) : {intercept}')

print(f'Optimal (Slope)     : {slope}')

**# 5. Prediction and plotting using linear equation**

# Calculate predicted weights using the linear equation

predicted\_weights = intercept + slope \* X

# Plot the data points and the regression line

plt.scatter(heights, weights, label="Original Data", marker='.', s=1, alpha=0.3, color='blue')

plt.plot(heights, predicted\_weights, color='red', label="Regression Line")

plt.xlabel("Height in (Feet)")

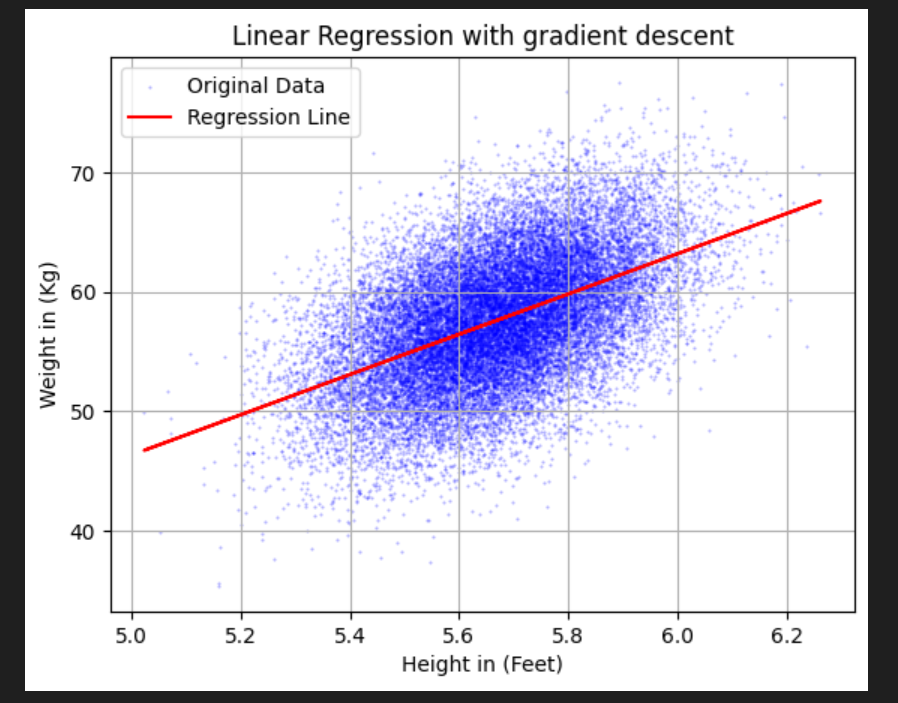
plt.ylabel("Weight in (Kg)")

plt.title("Linear Regression with gradient descent")

plt.legend()

plt.grid(True)

plt.show()

****

**DISCUSSION ON OUTPUT:**

1. **Optimal Intercept** (**-37.81)** : This value represents the y-intercept of the regression line. In the context of the data, it suggests that when height is zero (which doesn't make sense in real-life terms), the estimated weight is approximately -37.81 kilograms. This doesn't have a meaningful interpretation in this context and indicates a limitation of the linear regression model applied to this dataset.
2. **Optimal Slope** (**16.83)** : The slope represents the change in weight for a one-unit change in height. In this case, it indicates that, on average, for every additional foot in height, the estimated weight increases by approximately 16.83 kilograms.

This linear regression model aims to find the best-fitting line to predict weights based on heights. However, in this specific dataset, the interpretation of the intercept is not meaningful because it doesn't correspond to a realistic scenario. The slope coefficient, on the other hand, provides valuable information by indicating the average change in weight associated with changes in height.

**Practical :8**

**Realization of AND and OR functions using Hebbian Learning and display of decision boundary.**

**THEORETICAL DISCUSSION:**

Hebbian learning is a biological-inspired learning rule that strengthens synaptic connections between neurons when those neurons are activated simultaneously. To realize the AND and OR functions using Hebbian learning, we can model the problem as a simple perceptron with two input neurons and one output neuron.

* The AND function is “activated” when both the incoming inputs is “on”. In “neural” terms, the neuron fires when both the incoming signals are excitatory.

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* The OR function is “activated” when at least one of the incoming inputs is “on”. In “neural” terms, the neuron fires when at least one of the incoming signals is excitatory.

|  |  |  |
| --- | --- | --- |
| **x1** | **x2** | **y** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Hebbian learning adapts the weights based on co-activation, allowing the perceptron to learn.

**IMPLEMENTATION DETAILS/SOURCE CODE:**

**# 1. Importing Libraries**

import numpy as np

import matplotlib.pyplot as plt

**# 2. Defining utility functions**

# Function to plot decision boundary and the data points

def plot\_decision\_boundary(x, y, t, title):

    if (t == 0.4) :

        plt.axline((0, 4\*t), (4\*t, 0))

    else :

        plt.axline((0, t), (t, 0))

    plt.scatter(x[:, 0], x[:, 1], c=['green' if label == 1 else 'red' for label in y])

    plt.title(title)

    plt.show()

**# 3. Defining the inputs, target, learning rate and initializing weights for AND and OR**

# Define the input data and the target output

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y\_and = np.array([0, 0, 0, 1])

y\_or = np.array([0, 1, 1, 1])

# Define the learning rate

learning\_rate = 0.1

# Initialize the weights

weights\_and = np.zeros(2)

weights\_or = np.zeros(2)

**# 4. Updating weights using Hebbian learning**

# Train the model using Hebbian learning

for i in range(4):

    for x, y in zip(X, y\_and):

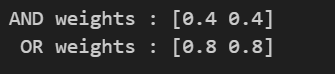
        weights\_and += learning\_rate \* y \* x

    for x, y in zip(X, y\_or):

        weights\_or += learning\_rate \* y \* x

print(f'AND weights : {weights\_and} \n OR weights : {weights\_or}')

Output:

****

**# 5. Display of decision boundaries and data points**

# AND

# Setting threshold value

t = 0.4

plot\_decision\_boundary(X, y\_and, t, 'AND function')

# OR

# Setting threshold value

t = 0.8

plot\_decision\_boundary(X, y\_or, t, 'OR function')

**A graph with red and green dots

Description automatically generatedA graph with a line

Description automatically generated**

**DISCUSSION ON OUTPUT:**

AND Weights: [0.4, 0.4]

OR Weights: [0.8, 0.8]

The weights obtained using Hebbian learning seem to approximate the behaviour of the AND and OR gates to some extent. However, the threshold for activation may not be ideal, and the network might not perfectly replicate the logical operations.

**Practical :9**

**Write a program to demonstrate competitive learning and display the results in a geometric format.**

**Theory:**

Competitive learning is a type of unsupervised learning where a set of neurons or units compete to represent input data. The most common competitive learning algorithm is known as the "Winner Takes All" (WTA) algorithm. Here all the output nodes try to compete with each other to represent the input pattern and the winner is declared according to the node having the most outputs and is given the output 1 while the rest are given 0.

There are a set of neurons with arbitrarily distributed weights and the activation function is applied to a subset of neurons. Only one neuron is active at a time. Only the winner has updated weights, the rest remain unchanged.

# Importing Libraries

import numpy as np

import matplotlib.pyplot as plt

# Defining required parameters

num\_clusters = np.random.randint(4, 8)

learning\_rate = 0.01

num\_iterations = 1000

input\_dimension = 2

input\_clusters = []

weight\_matrices = []

# Generate random input data for each cluster

for \_ in range(num\_clusters):

    # Generate random center in different quadrants

    quadrant = np.random.randint(1, 5)

    center = np.random.rand(input\_dimension) \* [-1, 1, 1, -1][quadrant - 1]

    # Generate random spread

    spread = np.random.rand(input\_dimension) \* 0.5

    # Generate random number of inputs

    num\_inputs = np.random.randint(4, 9)

    # Generate input data centered around the cluster center

    input\_data = np.random.rand(num\_inputs, input\_dimension) \* spread + center

    input\_clusters.append(input\_data)

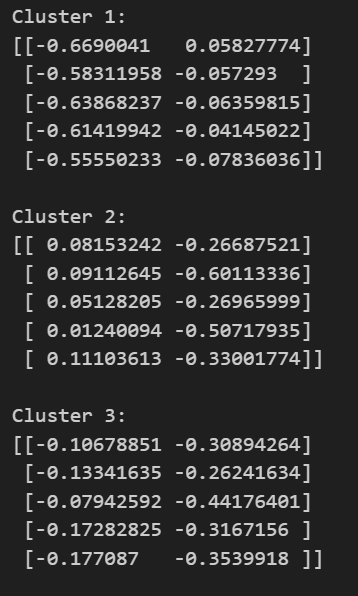
    # Initialize random weight matrix

    weight\_matrices.append(np.random.rand(1, input\_dimension))

# Print generated input data

for index, array in enumerate(input\_clusters):

    print(f'Cluster {index+1}:\n{array}\n')



# Application of competitive learning

for \_ in range(num\_iterations):

    for cluster\_idx, input\_data in enumerate(input\_clusters):

        for input\_vector in input\_data:

            # Calculate the weighted sum for each weight

            weighted\_sums = np.sum(input\_vector \* weight\_matrices[cluster\_idx], axis=1)

            # Choosing the winning neuron

            winning\_neuron\_idx = np.argmax(weighted\_sums)

            # Update the weights of the winning neuron using competitive learning rule

            weight\_matrices[cluster\_idx][winning\_neuron\_idx] += learning\_rate \* (input\_vector - weight\_matrices[cluster\_idx][winning\_neuron\_idx])

# Print the learned weights

for index, array in enumerate(weight\_matrices):

    print(f'Weight {index+1}:\n{array}\n')

A black screen with white text

Description automatically generated

# Plotting input data and final weight vectors

plt.figure(figsize=(10, 6))

# Plot input vectors

for cluster\_idx, input\_data in enumerate(input\_clusters):

    plt.scatter(input\_data[:, 0], input\_data[:, 1], label=f'Cluster {cluster\_idx + 1}', s=100)

# Plot weight vectors

for cluster\_idx, weights in enumerate(weight\_matrices):

    plt.quiver(0, 0, weights[0][0], weights[0][1], angles='xy', scale\_units='xy', scale=1, color=f'C{cluster\_idx}', width=0.005)

plt.xlabel('Dimension 1')

plt.ylabel('Dimension 2')

plt.title('Competitive Learning')

plt.legend()

plt.grid()

plt.show()

**Output:**

A graph with colored lines and dots

Description automatically generated

**Practical :10**

**Write a program to demonstrate Boltzmann Learning.**

**Theoretical Discussion:**

Boltzmann learning is statistical in nature, and is derived from the field of thermodynamics. It is similar to error-correction learning and is used during supervised training. In this algorithm, the state of each individual neuron, in addition to the system output, are taken into account. In this respect, the Boltzmann learning rule is significantly slower than the error-correction learning rule. Neural networks that use Boltzmann learning are called Boltzmann machines.

Boltzmann learning is similar to an error-correction learning rule, in that an error signal is used to train the system in each iteration. However, instead of a direct difference between the result value and the desired value, we take the difference between the probability distributions of the system.

**Source Code:**

import numpy as np

# Define the cost function to be minimized

def cost\_function(x):

    return x\*\*2 + 5 \* np.sin(x)

# Simulated Annealing (Boltzmann Learning) function

def simulated\_annealing(cost\_function, initial\_solution, temperature, cooling\_rate, iterations):

    current\_solution = initial\_solution

    current\_cost = cost\_function(current\_solution)

    best\_solution = current\_solution

    best\_cost = current\_cost

    for i in range(iterations):

        # Generate a random neighbor solution

        neighbor\_solution = current\_solution + np.random.uniform(-1, 1)

        neighbor\_cost = cost\_function(neighbor\_solution)

        # Calculate the change in cost

        cost\_change = neighbor\_cost - current\_cost

        # Accept the neighbor solution if it's better or with a certain probability

        if cost\_change < 0 or np.random.rand() < np.exp(-cost\_change / temperature):

            current\_solution = neighbor\_solution

            current\_cost = neighbor\_cost

        # Update the best solution if necessary

        if current\_cost < best\_cost:

            best\_solution = current\_solution

            best\_cost = current\_cost

        # Reduce the temperature

        temperature \*= cooling\_rate

    return best\_solution, best\_cost

if \_\_name\_\_ == "\_\_main\_\_":

    # Set initial parameters

    initial\_solution = np.random.uniform(-10, 10)

    initial\_temperature = 1000

    cooling\_rate = 0.99

    iterations = 1000

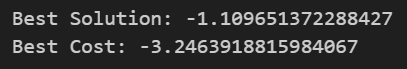
    # Find the minimum using Boltzmann Learning

    best\_solution, best\_cost = simulated\_annealing(cost\_function, initial\_solution, initial\_temperature, cooling\_rate, iterations)

    print("Best Solution:", best\_solution)

    print("Best Cost:", best\_cost)

**Output:**

****

**Practical 11 AIM:**

**Write a program to solve XOR problem using Multi-Layer Perceptron and draw the decision boundary.**

**Theoretical Discussion:**

The code uses the sigmoid activation function (σ) implemented as σ(value) to simulate the behaviour of neurons. The sigmoid function takes a weighted sum of inputs and produces an output between 0 and 1. The neural network consists of an input layer and a hidden layer with two neurons each. The mlp(x1, x2) function calculates the output of this network by computing the sigmoid activation of two hidden neurons and then passing their outputs to a final output neuron, effectively creating a small two-layer neural network. There's no explicit training in this code. Instead, the weights and biases are set manually within the mlp function to create a network that can solve the XOR problem. The XOR problem is a classic problem that's not linearly separable. The code attempts to approximate XOR logic using this simple network by setting specific weights and biases. The code uses matplotlib to create a 2D plot that shows the XOR input points (0,0), (0,1), (1,0), and (1,1) as green or red dots. The color indicates the output of the neural network when given those inputs. Green dots represent an output of 1, and red dots represent an output of 0. The line plt.axline((0, t), (t, 0)) is used to draw the decision boundary, separating the green and red dots. It represents the threshold (t) where the network transitions from outputting 0 to outputting 1. The plot visualizes how the neural network divides the input space into two regions, one for each class (0 or 1), showing the decision boundary that approximates the XOR problem.

**Source Code:**

import matplotlib.pyplot as plt

import numpy as np

import math

t = 0.5

def σ(value):

  return 1/(1 + np.exp(value)) >= t

def mlp(x1, x2):

   h1 = σ((1-x1) + x2)

   h2 = σ(x1 + (1-x2))

   return σ(h1 + h2)

# Define the input data and the target output

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y\_xor = []

for x in X:

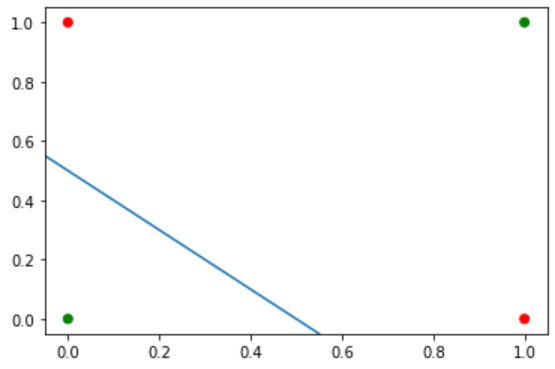
  y\_xor.append(mlp(x[0], x[1]))

plt.axline((0, t), (t, 0))

plt.scatter(X[:, 0], X[:, 1], c=['green' if label == 1 else 'red' for label in y\_xor])

plt.show()

**Output:**

****

**Practical 12**

**AIM: Write a Python program to implementation the Backpropagation Algorithm.**

**Code for the implementation of the problem statement:**

import numpy as np

def sigmoid(x):

  return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

  return sigmoid(x) \* (1 - sigmoid(x))

def backpropagation(input\_size, hidden\_layer\_sizes, output\_size, num\_iterations, X, Y,

learning\_rate):

    layer\_sizes = [input\_size] + hidden\_layer\_sizes + [output\_size]

    weights = [np.random.randn(layer\_sizes[i + 1], layer\_sizes[i]) for i in

    range(len(layer\_sizes) - 1)]

    biases = [np.zeros((layer\_sizes[i + 1], 1)) for i in range(len(layer\_sizes) - 1)]

    costs = []

    for i in range(num\_iterations):

        activations = [X]

        zs = []

    for j in range(len(weights)):

        z = np.dot(weights[j], activations[-1]) + biases[j]

        zs.append(z)

        a = sigmoid(z)

        activations.append(a)

        error = activations[-1] - Y

        cost = (1 / (2 \* X.shape[1])) \* np.sum(np.square(error))

        costs.append(cost)

        dZ = [activations[-1] - Y]

        dW = []

        db = []

    for j in range(len(weights) - 1, -1, -1):

        dW.insert(0, (1 / X.shape[1]) \* np.dot(dZ[0], activations[j - 1].T))

        db.insert(0, (1 / X.shape[1]) \* np.sum(dZ[0], axis=1, keepdims=True))

        if j > 0:

            dZ.insert(0, np.dot(weights[j].T, dZ[0]) \* sigmoid\_derivative(activations[j - 1]))

    for j in range(len(weights)):

        weights[j] -= learning\_rate \* dW[j]

        biases[j] -= learning\_rate \* db[j]

    return weights, biases, costs

input\_size = 2

hidden\_layer\_sizes = [2, 2]

output\_size = 1

num\_iterations = 2000

learning\_rate = 0.1

X = np.random.randn(input\_size, 1000)

Y = np.logical\_xor(X[0, :] > 0, X[1, :] > 0).astype(int).reshape(1, -1)

weights, biases, costs = backpropagation(input\_size, hidden\_layer\_sizes, output\_size,

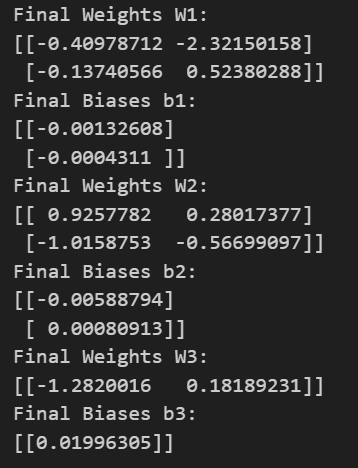
num\_iterations, X, Y, learning\_rate)

for j in range(len(weights)):

    print(f"Final Weights W{j + 1}:\n{weights[j]}")

    print(f"Final Biases b{j + 1}:\n{biases[j]}")

**Output:**

****