|  |
| --- |
| **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 |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**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)

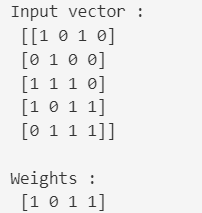
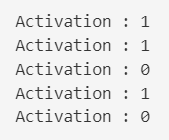
*# Checking activation for each input*

for i in ws2 :

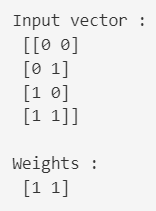
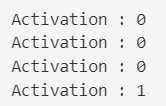
**print**("Activation : " + str(activation(t, i)))

**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 :

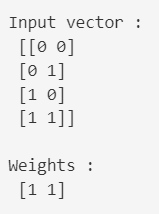
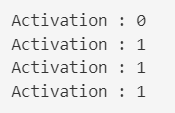
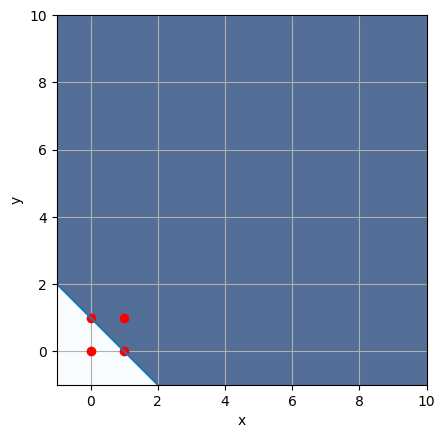
**print**("Activation : " + str(activation(t, i)))

*# Drawing decision boundary*

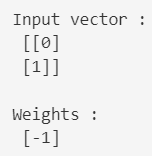
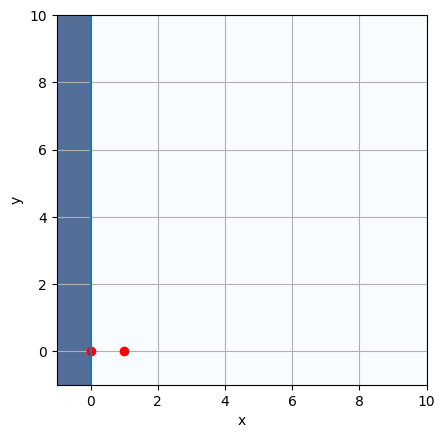
draw\_boundary(t, x3, w3)

**OUTPUT:**

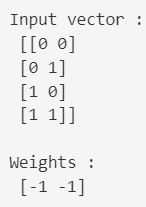
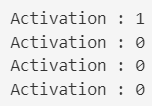
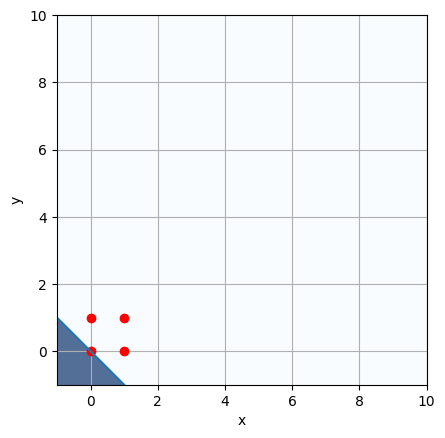
Realization of OR function:

****  ** **

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 :

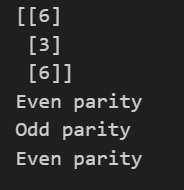
    if (parity(s) == 1) :

        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 :

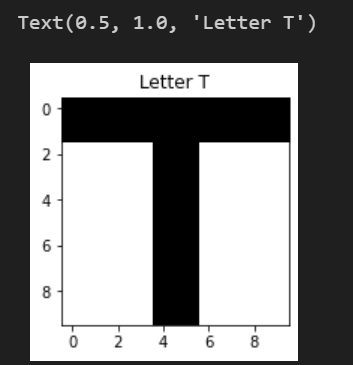
    cmap = matplotlib.colors.ListedColormap(["white", "red"])

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

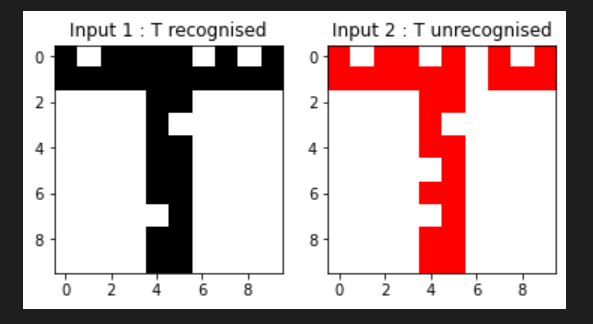
    plt.title("Input 2 : T unrecognised")

**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:

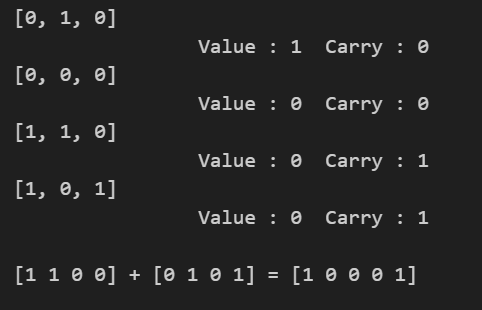
    res.append(carry)

# 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.