# Lab 1

## Problem Statement

Write a python program to create a neuron and predict its output using the threshold activation function.

## Theory

**Neurons**

A neuron is the building block of ANN, inspired from biological neurons in brain.

It takes multiple input, applies weights, computes a sum, and passes result through activation function to determine the output.

It consists of:

* Inputs: (x1, x2, … xn)
* Weights (w1, w2, …. Wn)
* Summation function (S = ∑(wi × xi)
* Activation function

**Threshold Activation Function**

Threshold Activation Function is the simplest activation function that decides whether the neuron should fire or not based on the computed sum.

This function is useful for binary classification problems (Yes/No, True/False, On/Off decisions).

## Program

class Neuron:

def \_\_init\_\_(self, weights, threshold):

self.weights = weights

self.threshold = threshold

def activate(self, inputs):

weighted\_sum = sum(w \* i for w, i in zip(self.weights, inputs))

return 1 if weighted\_sum >= self.threshold else 0

weights = list(map(float, input("Enter weights separated by spaces: ").split()))

threshold = float(input("Enter threshold value: "))

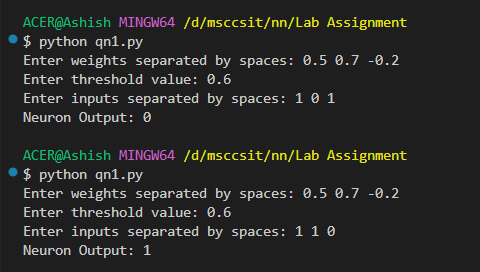
neuron = Neuron(weights, threshold)

inputs = list(map(float, input("Enter inputs separated by spaces: ").split()))

output = neuron.activate(inputs)

print("Neuron Output:", output)

## Outputs



## Calculation

* S=(0.5×1)+(0.7×0)+(−0.2×1)=0.5+0−0.2=0.3 (<0.6)
* S=(0.5×1)+(0.7×1)+(−0.2×0)=0.5+0.7+0=1.2 (>0.6)

# Lab 2

## Problem Statement

Write a python program to train AND Gate Using Perceptron Learning Algorithm.

## Theory

**Perceptron:**

The perceptron is the simplest form of a neural network used for the classifying linearly separable patterns. Patterns that lie on opposite sides of a hyperplane are called linearly separable patterns.

It is a type of artificial neuron that mimics how biological neurons work. It takes multiple inputs, applies weights, sums them, and then applies an activation function to decide the output.

A single-layer perceptron is useful for linearly separable problems, such as the AND, OR gates, but it cannot solve the XOR problem.

The summing node of the neural model computes a linear combination of the input. The resulting sum is applied to a hard limit activation function.

The neuron produces an output equal to 1 if the hard limiter input is positive, and -1 if it is negative.

The goal of the perceptron is to correctly classify the set of externally applied stimuli *x*1, *x*2, ..., *xm* into one of two classes, c1 or c2. The decision rule for the classification is to assign the point represented by the inputs *x*1, *x*2, ..., *xm* to class c1 if the perceptron output *y* is +1 and to class c2 if it is -1.

**Perceptron Learning Algorithm**

1. Initialize all weights and bias to zero
2. For each training vector s and target t perform steps 3 to 6
3. Set
4. Compute output using Hard limiter activation function as below



1. Adapt weights as:



1. Adapt bias as:



1. Test for Stopping Criteria

## Program:

import numpy as np

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.1, epochs=10):

self.learning\_rate = learning\_rate

self.epochs = epochs

self.weights = np.random.rand(2) # Initialize weights randomly

self.bias = np.random.rand(1) # Initialize bias randomly

def activation(self, x):

return 1 if x >= 0 else -1

def train(self, X, y):

for \_ in range(self.epochs):

for inputs, expected in zip(X, y):

weighted\_sum = np.dot(inputs, self.weights) + self.bias

output = self.activation(weighted\_sum)

error = expected - output

# Update weights and bias

self.weights += self.learning\_rate \* error \* np.array(inputs)

self.bias += self.learning\_rate \* error

def predict(self, inputs):

weighted\_sum = np.dot(inputs, self.weights) + self.bias

return self.activation(weighted\_sum)

# Training data for AND gate

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

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

# Train perceptron

perceptron = Perceptron(learning\_rate=0.1, epochs=10)

perceptron.train(X, y)

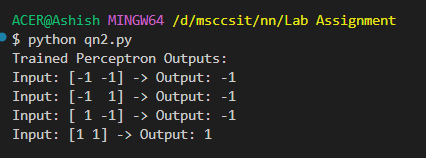
# Test perceptron

print("Trained Perceptron Outputs:")

for inputs in X:

print(f"Input: {inputs} -> Output: {perceptron.predict(inputs)}")

## Output



# Lab 3

## Problem Statement

Write a python program to implement Min-Max Scalar.

## Theory

Min-Max Scaling (also called Min-Max Normalization) is a feature scaling technique used in machine learning and data preprocessing to rescale numerical data into a specific range, typically [0, 1] or [-1, 1].

For a given value 𝑋, the Min-Max scaling formula is:

Where,

X=original data point

Xmin, Xmax = minimum and maximum data in dataset

Rmin, Rmax = Desired Range

Limitations of Min-Max Scaling

* Sensitive to Outliers: If data has extreme values, Min-Max scaling will compress most values into a narrow range.
* Not Robust: If new data comes in with different min/max values, the scaling needs to be recomputed.

## Program

import numpy as np

def min\_max\_scaler(data, feature\_range=(0, 1)):

min\_val, max\_val = feature\_range

min\_data = np.min(data)

max\_data = np.max(data)

if max\_data == min\_data:

return np.zeros\_like(data) if min\_val == 0 else np.full\_like(data, min\_val)

scaled\_data = (data - min\_data) / (max\_data - min\_data) \* (max\_val - min\_val) + min\_val

return scaled\_data

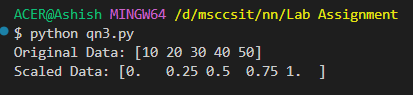
data = np.array([10, 20, 30, 40, 50])

scaled\_data = min\_max\_scaler(data, feature\_range=(0, 1))

print("Original Data:", data)

print("Scaled Data:", scaled\_data)

## Output



## Lab 4

## Problem Statement

Write a python program to implement Standard Scalar.

## Theory

The Standard Scaler is a preprocessing technique used in machine learning to standardize the features (or variables) of your dataset. The goal of standardization is to transform the data such that each feature has a mean of 0 and a standard deviation of 1, effectively putting all features on the same scale.

For the given value of x, the formula for Standard Scalar is,

Where, µ = mean, σ = standard deviation

## Program

import numpy as np

class StandardScaler:

def fit(self, X):

self.mean = np.mean(X, axis=0)

self.std = np.std(X, axis=0)

def transform(self, X):

return (X - self.mean) / self.std

def fit\_transform(self, X):

self.fit(X)

return self.transform(X)

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

data = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])

scaler = StandardScaler()

standardized\_data = scaler.fit\_transform(data)

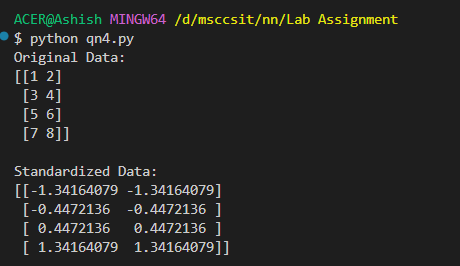
print("Original Data:")

print(data)

print("\nStandardized Data:")

print(standardized\_data)

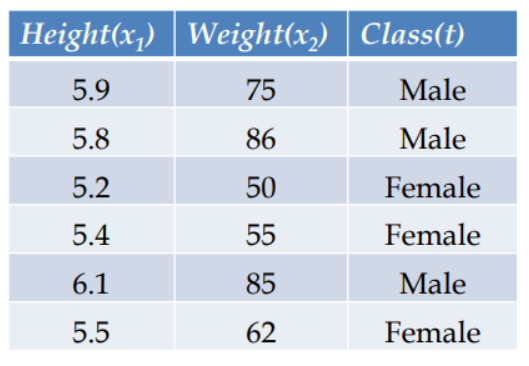
## Output



# Lab 5

## Problem Statement

Write a python program to train perceptron using given training set and predict class for the input (6,82) and (5.3,52)



## Theory

Let’s assume following value for given class labels.

Male = 1

Female = -1

We will use min max scaler to normalize the input value.

We will apply the perceptron learning algorithm to the normalized dataset to train the perceptron.

We will test with the given input data (6, 82) and (5.3, 52)

## Program

import numpy as np

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.1, epochs=10):

self.learning\_rate = learning\_rate

self.epochs = epochs

self.weights = np.random.rand(2)

self.bias = np.random.rand(1)

def activation(self, x):

return 1 if x >= 0 else -1

def train(self, X, y):

for \_ in range(self.epochs):

for inputs, expected in zip(X, y):

weighted\_sum = np.dot(inputs, self.weights) + self.bias

output = self.activation(weighted\_sum)

error = expected - output

# Update weights and bias

self.weights += self.learning\_rate \* error \* np.array(inputs)

self.bias += self.learning\_rate \* error

def predict(self, inputs):

weighted\_sum = np.dot(inputs, self.weights) + self.bias

return self.activation(weighted\_sum)

def min\_max\_scaler(data, feature\_range=(0, 1)):

min\_val, max\_val = feature\_range

min\_data = np.min(data, axis=0) # Find min of each column

max\_data = np.max(data, axis=0) # Find max of each column

# Avoid division by zero if min == max for a feature

if np.any(max\_data == min\_data):

return np.zeros\_like(data) if min\_val == 0 else np.full\_like(data, min\_val)

scaled\_data = (data - min\_data) / (max\_data - min\_data) \* (max\_val - min\_val) + min\_val

return scaled\_data

# Training data

data = np.array([

[5.9, 75],

[5.8, 86],

[5.2, 50],

[5.4, 55],

[6.1, 85],

[5.5, 62]

])

# Labels: 1 for Male, -1 for Female

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

# Normalize the data using Min-Max scaling (for all columns/features)

normalized\_data = min\_max\_scaler(data)

# Initialize and train the Perceptron

perceptron = Perceptron(learning\_rate=0.1, epochs=10)

perceptron.train(normalized\_data, labels)

# Test inputs

test\_inputs = np.array([

[6, 82],

[5.3, 52]

])

# Normalize the test inputs using the same scaling

normalized\_test\_inputs = min\_max\_scaler(test\_inputs)

# Make predictions for test inputs

predictions = [perceptron.predict(test\_input) for test\_input in normalized\_test\_inputs]

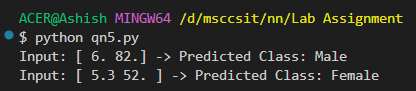
# Output predictions

for test\_input, prediction in zip(test\_inputs, predictions):

result = "Male" if prediction == 1 else "Female"

print(f"Input: {test\_input} -> Predicted Class: {result}")

## Output



# Lab 6

## Problem Statement

Implement Backpropagation algorithm to train an ANN of configuration 2X2X1 to achieve XOR function. (Use sigmoid and Tanh activation function). You have to implement and online as well as batch gradient descent.

## Theory

**Backpropagation Algorithm**

The backpropagation algorithm is a method used to train neural networks by adjusting their weights to minimize the error between the network's predictions and the actual outputs. It works by measuring the error at the output layer and then propagating this error backward through the network to adjust the weights accordingly.

**Steps:**

1. Initalize Weights and Biases Randomly
   1. Small random values for all weights (e.g., input to hidden, hidden to output).
   2. Biases often initialized to zero or small random values.
2. Forward Propagation
   1. Compute outputs layer-by-layer.
   2. For input x:
      1. Hidden Layer:
         1. 🡪 Linear Transformation
         2. 🡪 Activation Function
      2. Output Layer
   3. is the final predicted output
3. Compute Error
4. Backward Propagation: Here we adjust weights by how much they contributed to the error.
   1. Output Layer Gradient
      1. Compute gradient of the loss w.r.t. output
      2. Compute change needed for weights and biases
   2. Hidden Layer Gradient
      1. Backpropagate the error to hidden layer
      2. Compute update for input-to-hidden weights
5. Update Weights and Biases
   1. Updates using gradient descent rules
6. Repeat for many epochs

**Types of Training**

1. Online Gradient Descent: Also known as Stochastic Gradient Descent, Update weights after every single input example.
2. Batch Gradient Descent: Calculate updates after seeing the entire dataset once.

## Program

import numpy as np

#XOR DATASET

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

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

#ACTIVATION FUNCTION

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

def tanh(x):

return np.tanh(x)

def tanh\_derivative(x):

return 1 - np.tanh(x)\*\*2

#ALGORITHM

def train\_xor(X, Y, activation="sigmoid", epochs=10000, learning\_rate=0.1, batch\_mode=False):

#Initialize weights and biases

np.random.seed(42)

w1 = np.random.rand(2, 2) #INPUT TO HIDDEN

b1 = np.zeros((1, 2))

w2 = np.random.rand(2, 1) #HIDDEN TO OUTPUT

b2 = np.zeros((1, 1))

#Choose activation

if activation == "sigmoid":

activation\_function = sigmoid

activation\_derivative = sigmoid\_derivative

elif activation == "tanh":

activation\_function = tanh

activation\_derivative = tanh\_derivative

else:

raise ValueError("Invalid activation function")

#Train

for epoch in range(epochs):

if batch\_mode:

#forward pass

z1 = np.dot(X, w1) + b1

a1 = activation\_function(z1)

z2 = np.dot(a1, w2) + b2

a2 = activation\_function(z2)

#backpropagation

error = Y - a2

dz2 = error \* activation\_derivative(a2)

dw2 = np.dot(a1.T, dz2)

db2 = np.sum(dz2, axis=0, keepdims=True)

dz1 = np.dot(dz2, w2.T) \* activation\_derivative(a1)

dw1 = np.dot(X.T, dz1)

db1 = np.sum(dz1, axis=0)

#updates

w1 += learning\_rate \* dw1

b1 += learning\_rate \* db1

w2 += learning\_rate \* dw2

b2 += learning\_rate \* db2

else:

for i in range(len(X)):

x = X[i:i+1]

y = Y[i:i+1]

#forward pass

z1 = np.dot(x, w1) + b1

a1 = activation\_function(z1)

z2 = np.dot(a1, w2) + b2

a2 = activation\_function(z2)

#backpropagation

error = y - a2

dz2 = error \* activation\_derivative(a2)

dw2 = np.dot(a1.T, dz2)

db2 = dz2

dz1 = np.dot(dz2, w2.T) \* activation\_derivative(a1)

dw1 = np.dot(x.T, dz1)

db1 = dz1

#updates

w1 += learning\_rate \* dw1

b1 += learning\_rate \* db1

w2 += learning\_rate \* dw2

b2 += learning\_rate \* db2

if epoch % 1000 == 0:

loss = np.mean((Y - a2) \*\* 2)

print(f"Epoch {epoch}, Loss: {loss: .4f}")

return w1, b1, w2, b2

#EVALUATION

def evaluate\_xor(X, w1, b1, w2, b2, activation="sigmoid"):

if activation == "sigmoid":

activation\_function = sigmoid

elif activation == "tanh":

activation\_function = tanh

a1 = activation\_function(np.dot(X, w1) + b1)

a2 = activation\_function(np.dot(a1, w2) + b2)

return a2

# MAIN

def main():

print("Train XOR ANN (2x2x1)")

activation = input("Choose activation function (sigmoid/tanh): ").strip().lower()

mode = input("Choose training mode (batch/online): ").strip().lower()

epochs = int(input("Number of training epochs [default: 10000]: ") or 10000)

lr = float(input("Learning rate [default: 0.1]: ") or 0.1)

batch\_mode = True if mode == "batch" else False

print("\nTraining started...\n")

w1, b1, w2, b2 = train\_xor(X, Y, activation=activation, epochs=epochs, learning\_rate=lr, batch\_mode=batch\_mode)

preds = evaluate\_xor(X, w1, b1, w2, b2, activation=activation)

print("Input\tExpected\tPredicted")

for i in range(len(X)):

x1, x2 = X[i]

expected = Y[i][0]

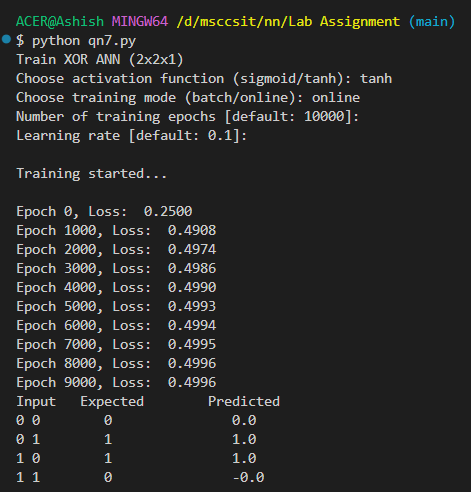
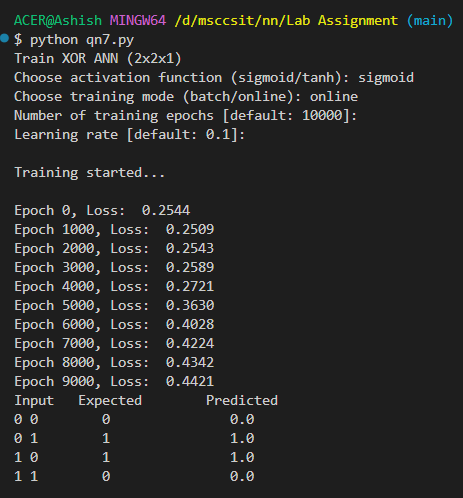
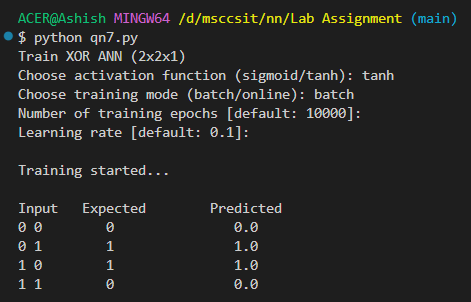
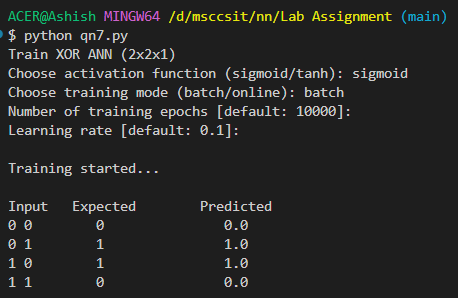
predicted = np.round(preds[i][0])

print(f"{x1} {x2}\t {expected}\t\t {predicted}")

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

main()

Output

1. With Online Gradient Descent
   1. Tanh Activation Function
      1. 
   2. Sigmoid Activation Function
      1. 
2. With Batch Gradient Descent
   1. Tanh Activation Function
      1. 
   2. Sigmoid Activation Function
      1. 

# Lab 7

## Problem Statement

Implement Backpropagation algorithm to train an ANN of configuration 3X2X2X1 to achieve majority function with 3-bit data. Output of the network must be 1 when there are two or more 1’s in the data. (Use sigmoid and Tanh activation function). You have to implement and online as well as batch gradient descent.

## Theory

Majority Function Dataset:

|  |  |  |  |
| --- | --- | --- | --- |
| X1 | X2 | X3 | Output |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |

Program:

import numpy as np

# Majority Function Dataset

X = np.array([

[0, 0, 0],

[0, 0, 1],

[0, 1, 0],

[0, 1, 1],

[1, 0, 0],

[1, 0, 1],

[1, 1, 0],

[1, 1, 1]

])

Y = np.array([

[0],

[0],

[0],

[1],

[0],

[1],

[1],

[1]

])

# Activation Functions

def sigmoid(x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(x):

return x \* (1 - x)

def tanh(x):

return np.tanh(x)

def tanh\_derivative(x):

return 1 - np.tanh(x)\*\*2

# Training Function

def train\_majority(X, Y, activation="sigmoid", epochs=10000, learning\_rate=0.1, batch\_mode=False):

np.random.seed(42)

w1 = np.random.randn(3, 2)

b1 = np.zeros((1, 2))

w2 = np.random.randn(2, 2)

b2 = np.zeros((1, 2))

w3 = np.random.randn(2, 1)

b3 = np.zeros((1, 1))

if activation == "sigmoid":

act = sigmoid

act\_derivative = sigmoid\_derivative

elif activation == "tanh":

act = tanh

act\_derivative = tanh\_derivative

else:

raise ValueError("Invalid activation function")

for epoch in range(epochs):

if batch\_mode:

z1 = np.dot(X, w1) + b1

a1 = act(z1)

z2 = np.dot(a1, w2) + b2

a2 = act(z2)

z3 = np.dot(a2, w3) + b3

a3 = act(z3)

error = Y - a3

dz3 = error \* act\_derivative(a3)

dw3 = np.dot(a2.T, dz3)

db3 = np.sum(dz3, axis=0, keepdims=True)

dz2 = np.dot(dz3, w3.T) \* act\_derivative(a2)

dw2 = np.dot(a1.T, dz2)

db2 = np.sum(dz2, axis=0, keepdims=True)

dz1 = np.dot(dz2, w2.T) \* act\_derivative(a1)

dw1 = np.dot(X.T, dz1)

db1 = np.sum(dz1, axis=0, keepdims=True)

w1 += learning\_rate \* dw1

b1 += learning\_rate \* db1

w2 += learning\_rate \* dw2

b2 += learning\_rate \* db2

w3 += learning\_rate \* dw3

b3 += learning\_rate \* db3

else:

for i in range(len(X)):

x = X[i:i+1]

y = Y[i:i+1]

z1 = np.dot(x, w1) + b1

a1 = act(z1)

z2 = np.dot(a1, w2) + b2

a2 = act(z2)

z3 = np.dot(a2, w3) + b3

a3 = act(z3)

error = y - a3

dz3 = error \* act\_derivative(a3)

dw3 = np.dot(a2.T, dz3)

db3 = dz3

dz2 = np.dot(dz3, w3.T) \* act\_derivative(a2)

dw2 = np.dot(a1.T, dz2)

db2 = dz2

dz1 = np.dot(dz2, w2.T) \* act\_derivative(a1)

dw1 = np.dot(x.T, dz1)

db1 = dz1

w1 += learning\_rate \* dw1

b1 += learning\_rate \* db1

w2 += learning\_rate \* dw2

b2 += learning\_rate \* db2

w3 += learning\_rate \* dw3

b3 += learning\_rate \* db3

if epoch % 1000 == 0:

loss = np.mean((Y - a3) \*\* 2)

print(f"Epoch {epoch}, Loss: {loss:.4f}")

return w1, b1, w2, b2, w3, b3

# Evaluation Function

def evaluate\_majority(X, w1, b1, w2, b2, w3, b3, activation="sigmoid"):

if activation == "sigmoid":

act = sigmoid

elif activation == "tanh":

act = tanh

a1 = act(np.dot(X, w1) + b1)

a2 = act(np.dot(a1, w2) + b2)

a3 = act(np.dot(a2, w3) + b3)

return a3

# User chooses Activation

print("Choose Activation Function:")

print("1. Sigmoid")

print("2. Tanh")

activation\_choice = input("Enter 1 or 2: ").strip()

if activation\_choice == "1":

activation\_choice = "sigmoid"

elif activation\_choice == "2":

activation\_choice = "tanh"

else:

print("Invalid choice. Defaulting to Sigmoid.")

activation\_choice = "sigmoid"

# User chooses Training Mode

print("\nChoose Training Mode:")

print("1. Batch Gradient Descent")

print("2. Online Gradient Descent")

mode\_choice = input("Enter 1 or 2: ").strip()

if mode\_choice == "1":

batch\_mode = True

elif mode\_choice == "2":

batch\_mode = False

else:

print("Invalid choice. Defaulting to Batch mode.")

batch\_mode = True

# Training

print(f"\nTraining using {activation\_choice.upper()} activation and {'BATCH' if batch\_mode else 'ONLINE'} mode...\n")

w1, b1, w2, b2, w3, b3 = train\_majority(X, Y, activation=activation\_choice, epochs=10000, learning\_rate=0.1, batch\_mode=batch\_mode)

# Evaluate

preds = evaluate\_majority(X, w1, b1, w2, b2, w3, b3, activation=activation\_choice)

# Show final predictions

print("\nFinal Predictions:")

print("Input\t\tExpected\tPredicted")

for i in range(len(X)):

x1, x2, x3 = X[i]

expected = Y[i][0]

predicted = np.round(preds[i][0])

print(f"{x1} {x2} {x3}\t\t {expected}\t\t {predicted}")

while True:

user\_input = input("\nEnter 3 bits separated by spaces (e.g., '1 0 1') or type 'exit' to quit: ").strip()

if user\_input.lower() == 'exit':

print("Goodbye!")

break

try:

bits = list(map(int, user\_input.split()))

if len(bits) != 3 or any(b not in (0, 1) for b in bits):

print("Invalid input! Please enter exactly three 0 or 1 values.")

continue

bits\_array = np.array(bits).reshape(1, -1)

user\_pred = evaluate\_majority(bits\_array, w1, b1, w2, b2, w3, b3, activation=activation\_choice)

user\_pred\_binary = np.round(user\_pred[0][0])

print(f"Predicted Output: {int(user\_pred\_binary)}")

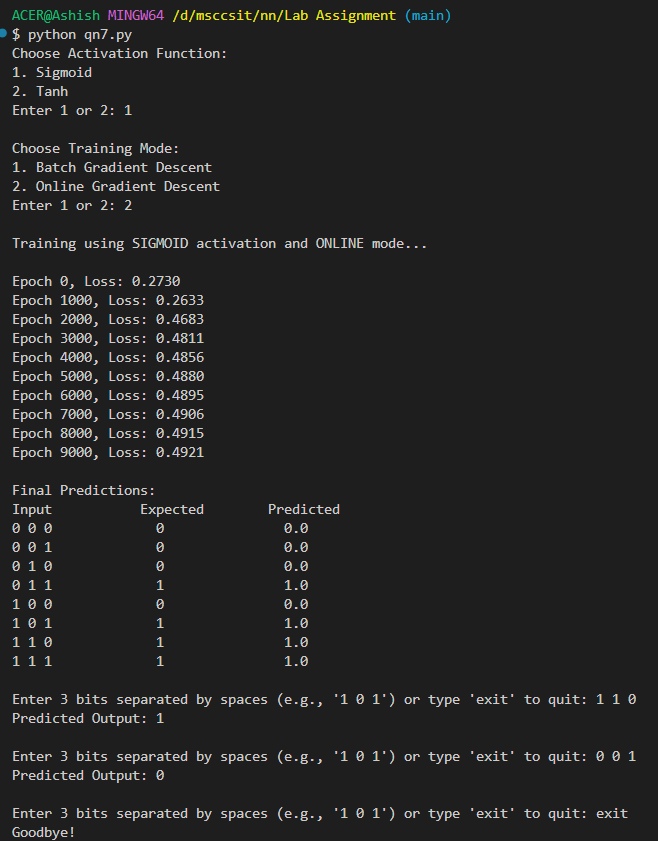
except Exception as e:

print("Error:", e)

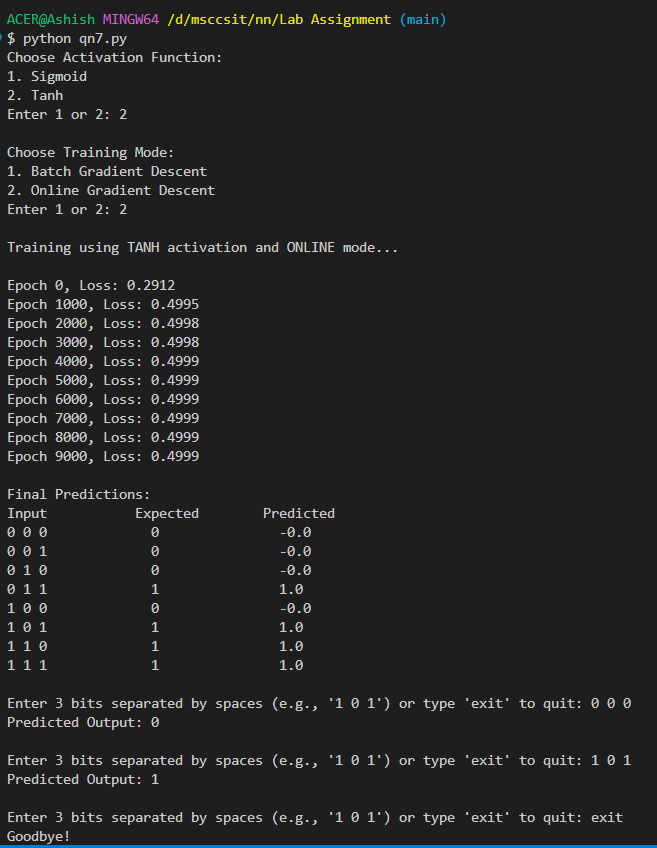
continue

## Output

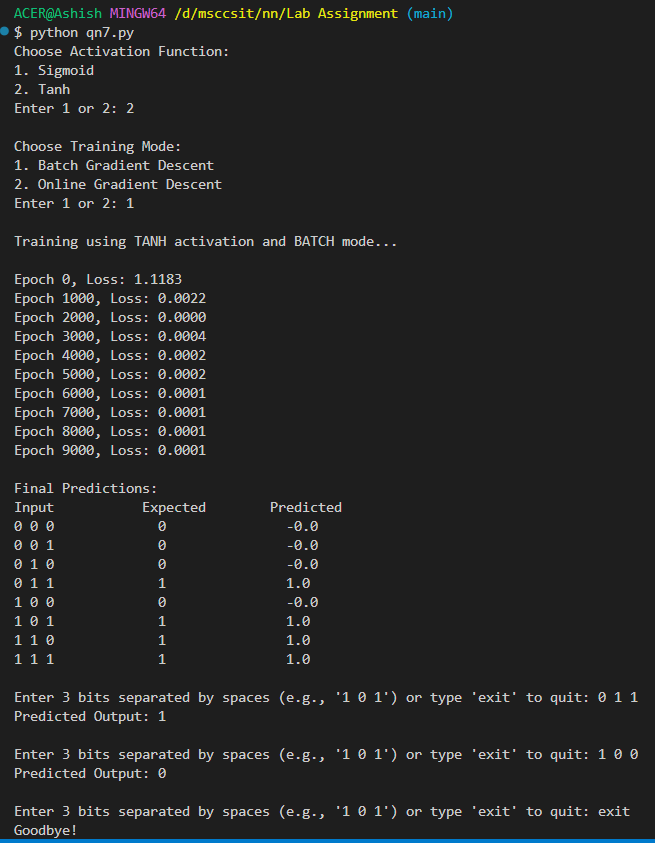
1. With Online Gradient Descent
   1. Sigmoid Activation Function



* 1. Tanh Activation Function



1. With Batch Gradient Descent
   1. Tanh Activation Function



* 1. Sigmoid Activation Function

