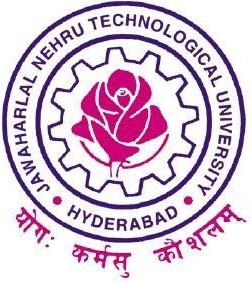
**J.N.T.U.H. UNIVERSITY COLLEGE OF ENGINEERING, SCIENCE AND TECHNOLOGY HYDERABAD**

**KUKATPALLY, HYDERABAD – 500 085**



**CERTIFICATE**

This is to certify that **NANDINI MAHARAJ** of B.Tech III year II Semester bearing the Hall-Ticket number **21011A0565** has fulfilled his/her DEEP LEARNING LAB record for the academic year 2023-2024.

Signature of the Head of the Department Signature of the staff member

Date of Examination:

Internal Examiner External Examiner

**TABLE OF CONTENTS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sno. | Name of the experiment | Date | Page no. | Sign |
| 1. | Logic Gates using Perceptron   1. OR 2. AND 3. NAND 4. NOR 5. NOT 6. XOR |  |  |  |
| 2. | ADALINE |  |  |  |
| 3. | MADALINE |  |  |  |
| 4. | Image Classification **using CNN on** **MNIST dataset** |  |  |  |
| 5. | Image Classification **using CNN on CIFAR 10 dataset** |  |  |  |

1. Logic Gates using Perceptron – OR gate

def activate(x):

    return 1 if x >= 0 else 0

def perceptron(inputs):

    w1, w2, b = 0, 0, 0

    desired\_outputs = [0, 1, 1, 1]

    learning\_rate = 0.1

    epochs = 100

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            A, B = inputs[i]

            target\_output = desired\_outputs[i]

            output = activate(w1 \* A + w2 \* B + b)

            error = target\_output - output

            w1 += learning\_rate \* error \* A

            w2 += learning\_rate \* error \* B

            b += learning\_rate \* error

            total\_error += abs(error)

        if total\_error == 0:

            break

    if total\_error == 0:

        return w1, w2, b

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

w1, w2, b = perceptron(inputs)

print("Weights:", w1, w2)

print("Bias:", b)

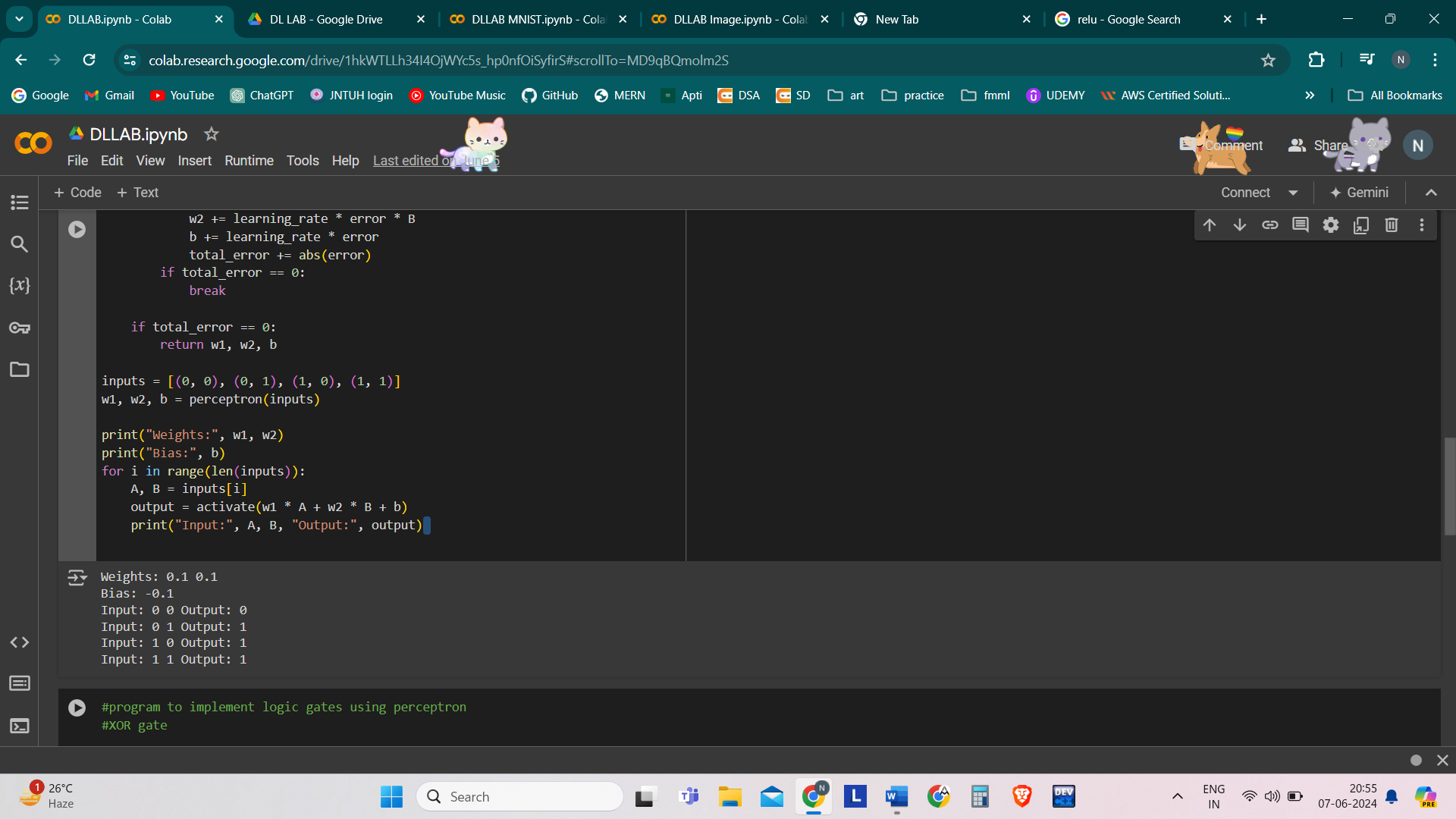
for i in range(len(inputs)):

    A, B = inputs[i]

    output = activate(w1 \* A + w2 \* B + b)

    print("Input:", A, B, "Output:", output)

**OUTPUT:**

****

1. Logic Gates using Perceptron – AND gate

def activate(x):

    return 1 if x >= 0 else 0

def perceptron(inputs):

    w1, w2, b = 0, 0, 0

    desired\_outputs = [0, 0, 0, 1]

    learning\_rate = 0.1

    epochs = 100

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            A, B = inputs[i]

            target\_output = desired\_outputs[i]

            output = activate(w1 \* A + w2 \* B + b)

            error = target\_output - output

            w1 += learning\_rate \* error \* A

            w2 += learning\_rate \* error \* B

            b += learning\_rate \* error

            total\_error += abs(error)

        if total\_error == 0:

            break

    if total\_error == 0:

        return w1, w2, b

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

w1, w2, b = perceptron(inputs)

print("Weights:", w1, w2)

print("Bias:", b)

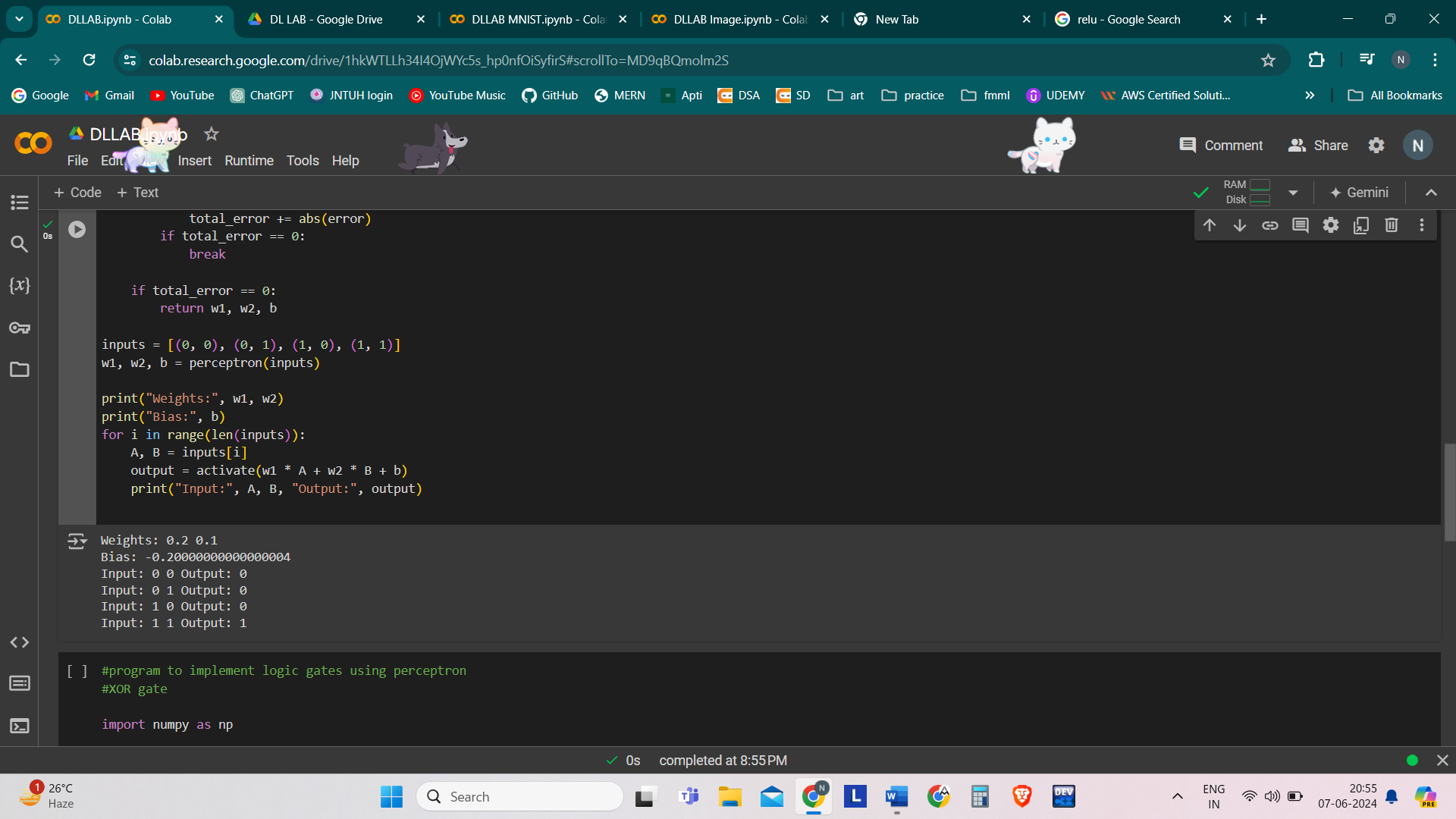
for i in range(len(inputs)):

    A, B = inputs[i]

    output = activate(w1 \* A + w2 \* B + b)

    print("Input:", A, B, "Output:", output)

**OUTPUT:**



1. Logic Gates using Perceptron – NOR gate

def activate(x):

    return 1 if x >= 0 else 0

def perceptron(inputs):

    w1, w2, b = 0, 0, 0

    desired\_outputs = [1, 0, 0, 0]

    learning\_rate = 0.1

    epochs = 100

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            A, B = inputs[i]

            target\_output = desired\_outputs[i]

            output = activate(w1 \* A + w2 \* B + b)

            error = target\_output - output

            w1 += learning\_rate \* error \* A

            w2 += learning\_rate \* error \* B

            b += learning\_rate \* error

            total\_error += abs(error)

        if total\_error == 0:

            break

    if total\_error == 0:

        return w1, w2, b

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

w1, w2, b = perceptron(inputs)

print("Weights:", w1, w2)

print("Bias:", b)

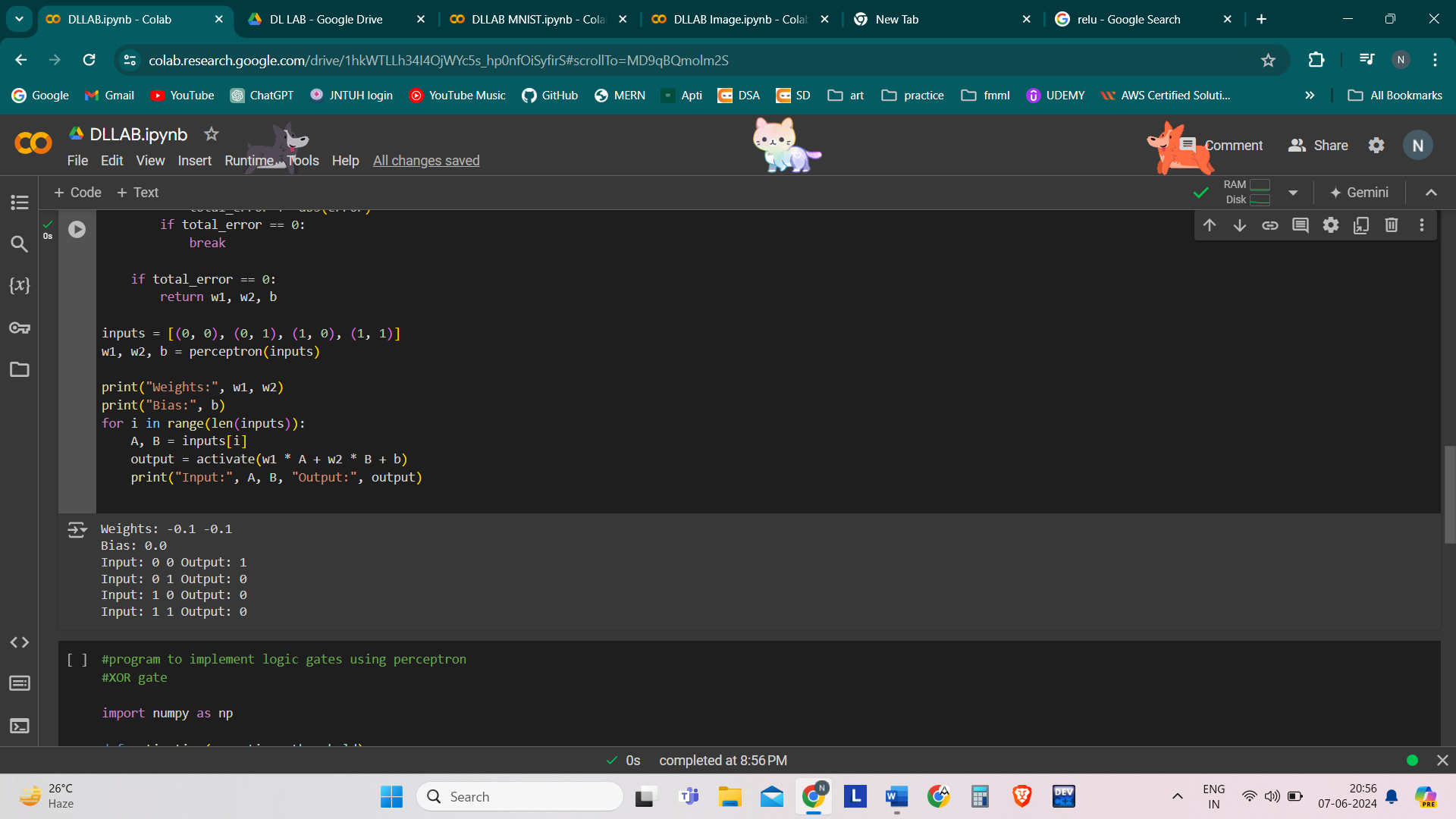
for i in range(len(inputs)):

    A, B = inputs[i]

    output = activate(w1 \* A + w2 \* B + b)

    print("Input:", A, B, "Output:", output)

**OUTPUT:**

****

1. Logic Gates using Perceptron – NAND gate

def activate(x):

    return 1 if x >= 0 else 0

def perceptron(inputs):

    w1, w2, b = 0, 0, 0

    desired\_outputs = [0, 1, 1, 1] #[0, 0, 0, 1] #[1, 0, 0, 0] #[1, 1, 1, 0]

    learning\_rate = 0.1

    epochs = 100

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            A, B = inputs[i]

            target\_output = desired\_outputs[i]

            output = activate(w1 \* A + w2 \* B + b)

            error = target\_output - output

            w1 += learning\_rate \* error \* A

            w2 += learning\_rate \* error \* B

            b += learning\_rate \* error

            total\_error += abs(error)

        if total\_error == 0:

            break

    if total\_error == 0:

        return w1, w2, b

inputs = [(0, 0), (0, 1), (1, 0), (1, 1)]

w1, w2, b = perceptron(inputs)

print("Weights:", w1, w2)

print("Bias:", b)

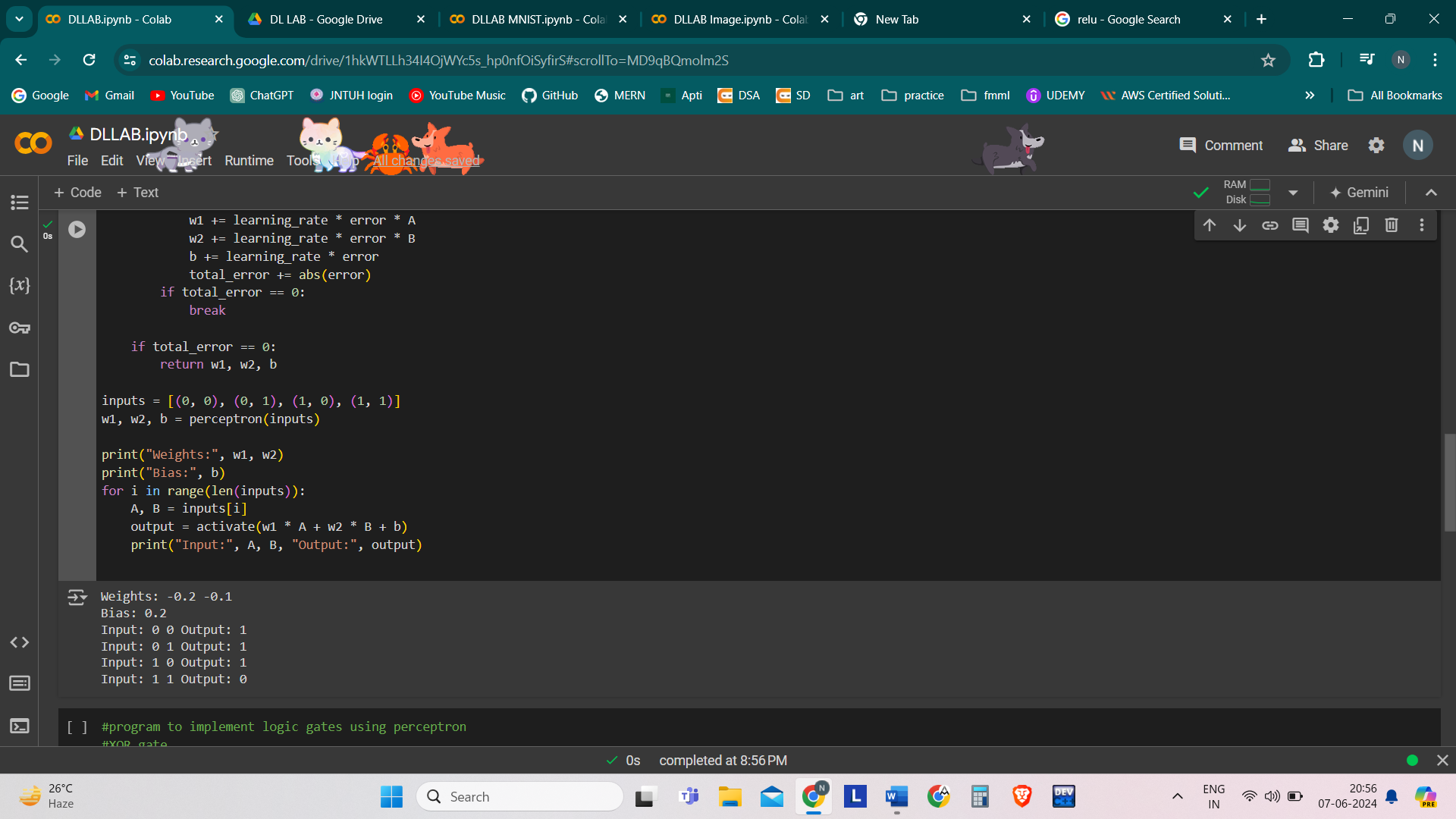
for i in range(len(inputs)):

    A, B = inputs[i]

    output = activate(w1 \* A + w2 \* B + b)

    print("Input:", A, B, "Output:", output)

**OUTPUT:**

****

1. Logic Gates using Perceptron – NOT gate

def activate(x):

    return 1 if x >= 0 else 0

def perceptron(inputs):

    w1, b = 0, -1

    desired\_outputs = [1, 0]

    learning\_rate = 0.1

    epochs = 100

    for epoch in range(epochs):

        total\_error = 0

        for i in range(len(inputs)):

            A = inputs[i]

            target\_output = desired\_outputs[i]

            output = activate(w1 \* A + b)

            error = target\_output - output

            w1 += learning\_rate \* error \* A

            b += learning\_rate \* error

            total\_error += abs(error)

        if total\_error == 0:

            break

    if total\_error == 0:

        return w1, b

inputs = [0, 1]

w1, b = perceptron(inputs)

print("NOT Gate Output:")

print("Weight:", w1)

print("Bias:", b)

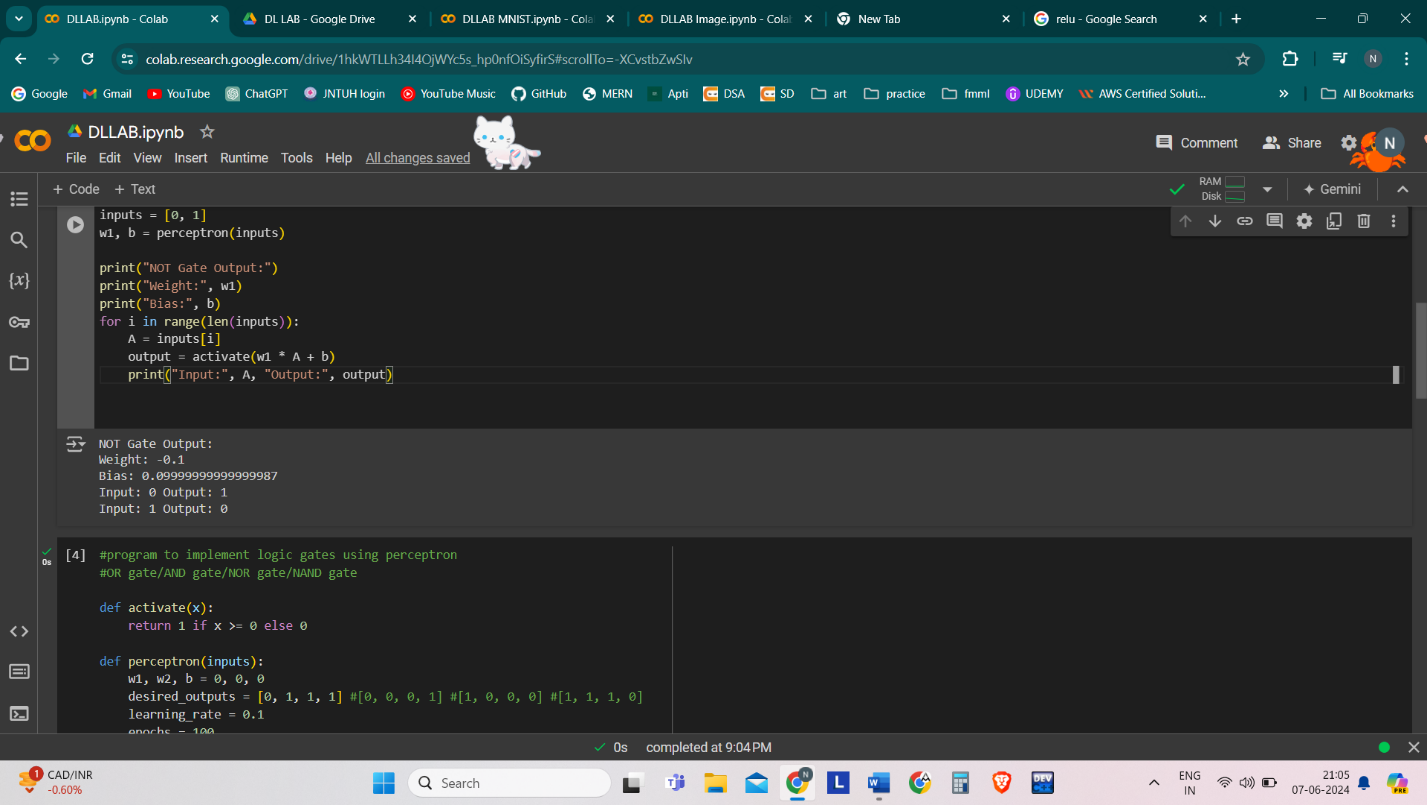
for i in range(len(inputs)):

    A = inputs[i]

    output = activate(w1 \* A + b)

    print("Input:", A, "Output:", output)

**OUTPUT:**

****

1. Logic Gates using Perceptron – XOR gate

import numpy as np

# define Unit Step Function

def unitStep(v):

  if v >= 0:

    return 1

  else:

    return 0

# design Perceptron Model

def perceptronModel(x, w, b):

  v = np.dot(w, x) + b

  y = unitStep(v)

  return y

# NOT Logic Function

# wNOT = -1, bNOT = 0.5

def NOT\_logicFunction(x):

  wNOT = -1

  bNOT = 0.5

  return perceptronModel(x, wNOT, bNOT)

# AND Logic Function

# here w1 = wAND1 = 1,

# w2 = wAND2 = 1, bAND = -1.5

def AND\_logicFunction(x):

  w = np.array([1, 1])

  bAND = -1.5

  return perceptronModel(x, w, bAND)

# OR Logic Function

# w1 = 1, w2 = 1, bOR = -0.5

def OR\_logicFunction(x):

  w = np.array([1, 1])

  bOR = -0.5

  return perceptronModel(x, w, bOR)

# XOR Logic Function

# with AND, OR and NOT

# function calls in sequence

def XOR\_logicFunction(x):

  y1 = AND\_logicFunction(x)

  y2 = OR\_logicFunction(x)

  y3 = NOT\_logicFunction(y1)

  final\_x = np.array([y2, y3])

  finalOutput = AND\_logicFunction(final\_x)

  return finalOutput

# testing the Perceptron Model

test1 = np.array([0, 1])

test2 = np.array([1, 1])

test3 = np.array([0, 0])

test4 = np.array([1, 0])

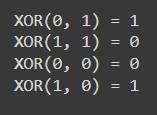
print("XOR({}, {}) = {}".format(0, 1, XOR\_logicFunction(test1)))

print("XOR({}, {}) = {}".format(1, 1, XOR\_logicFunction(test2)))

print("XOR({}, {}) = {}".format(0, 0, XOR\_logicFunction(test3)))

print("XOR({}, {}) = {}".format(1, 0, XOR\_logicFunction(test4)))

**OUTPUT:**



1. ADALINE Neural Network

import numpy as np

def Adaline(Input, Target, lr=0.2, stop=0.001):

weight = np.random.random(Input.shape[1])

bias = np.random.random(1)[0] # Extract scalar from array

Error = [stop + 1]

# Check the stop condition for the network

while Error[-1] > stop or Error[-1] - Error[-2] > 0.0001:

error = []

for i in range(Input.shape[0]):

Y\_input = np.dot(weight, Input[i]) + bias

# Update the weight

for j in range(Input.shape[1]):

weight[j] = weight[j] + lr \* (Target[i] - Y\_input) \* Input[i][j]

# Update the bias

bias = bias + lr \* (Target[i] - Y\_input)

# Store squared error value

error.append((Target[i] - Y\_input) \*\* 2)

# Store sum of squared errors

Error.append(sum(error))

return weight, bias

# Input dataset

x = np.array([[1.0, 1.0, 1.0],

[1.0, -1.0, 1.0],

[-1.0, 1.0, 1.0],

[-1.0, -1.0, -1.0]])

# Target values

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

# Train the Adaline model

w, b = Adaline(x, t, lr=0.2, stop=0.001)

# Print the final weights and bias

print('Weights:', w)

print('Bias:', b)

# Predict outputs

predicted\_outputs = []

for i in range(x.shape[0]):

predicted\_output = np.dot(w, x[i]) + b

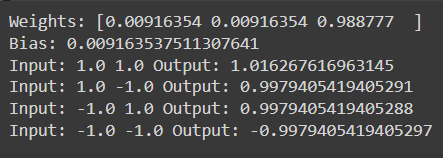
predicted\_outputs.append(predicted\_output)

# Display inputs and predicted outputs

for i in range(x.shape[0]):

print("Input:", x[i][0], x[i][1], "Output:", predicted\_outputs[i])

**OUTPUT:**



1. MADALINE Neural Network

import numpy as np

# Activation function

def activation\_fn(z):

return 1 if z >= 0 else -1

def Madaline(Input, Target, lr, epoch):

weight = np.random.random((Input.shape[1], Input.shape[1]))

bias = np.random.random(Input.shape[1])

w = np.array([0.5 for \_ in range(weight.shape[1])])

b = 0.5

k = 0

while k < epoch:

error = []

z\_input = np.zeros(bias.shape[0])

z = np.zeros(bias.shape[0])

for i in range(Input.shape[0]):

for j in range(Input.shape[1]):

z\_input[j] = sum(weight[j] \* Input[i]) + bias[j]

z[j] = activation\_fn(z\_input[j])

y\_input = sum(z \* w) + b

y = activation\_fn(y\_input)

# Update the weight & bias

if y != Target[i]:

for j in range(weight.shape[1]):

weight[j] = weight[j] + lr \* (Target[i] - z\_input[j]) \* Input[i][j]

bias[j] = bias[j] + lr \* (Target[i] - z\_input[j])

# Store squared error value

error.append((Target[i] - y\_input) \*\* 2)

# Compute sum of square error

Error = sum(error)

k += 1

return weight, bias

# Prediction function

def prediction(X, w, b):

y = []

for i in range(X.shape[0]):

x = X[i]

z1 = x \* w

z\_1 = []

for j in range(z1.shape[1]):

z\_1.append(activation\_fn(sum(z1[j]) + b[j]))

y\_in = sum(np.array(z\_1) \* np.array([0.5 for \_ in range(w.shape[1])])) + 0.5

y.append(activation\_fn(y\_in))

return y

# Input dataset

x = np.array([[1.0, 1.0, 1.0],

[1.0, -1.0, 1.0],

[-1.0, 1.0, 1.0],

[-1.0, -1.0, -1.0]])

# Target values

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

# Train the MADALINE model

w, b = Madaline(x, t, lr=0.0001, epoch=3)

# Print the final weights and bias

print('Weights:', w)

print('Bias:', b)

# Predict outputs

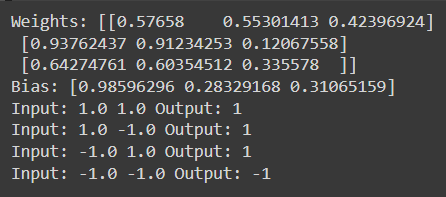
predicted\_outputs = prediction(x, w, b)

# Display inputs and predicted outputs

for i in range(x.shape[0]):

print("Input:", x[i][0], x[i][1], "Output:", predicted\_outputs[i])

**OUTPUT:**



1. Image Classification using MNIST dataset

import tensorflow as tf

import keras

from tensorflow.keras import datasets, layers, models

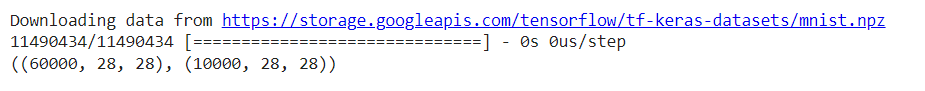
import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

(X\_train, y\_train), (X\_test, y\_test) = keras.datasets.mnist.load\_data()

X\_train.shape, X\_test.shape



#Normalisation

X\_train=X\_train/255

X\_test=X\_test/255

cnn=models.Sequential([

    #cnn

    layers.Conv2D(filters=32,kernel\_size=(3,3),activation='relu',input\_shape=(28,28,1)),

    layers.MaxPooling2D((2,2)),

    layers.Conv2D(filters=64,kernel\_size=(3,3),activation='relu'),

    layers.MaxPooling2D((2,2)),

    #dense

    layers.Flatten(),

    layers.Dense(50, activation='relu'),

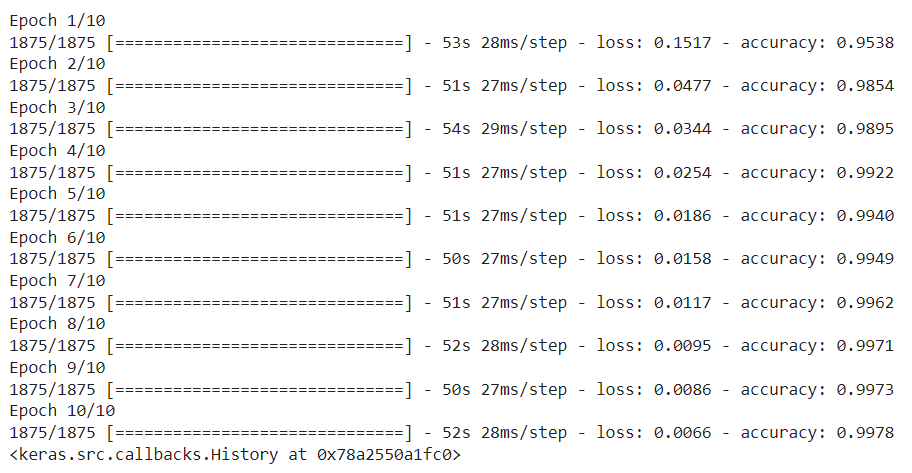
    layers.Dense(10, activation='softmax')

])

cnn.compile(optimizer='adam',

            loss='sparse\_categorical\_crossentropy',

            metrics=['accuracy'])

cnn.fit(X\_train, y\_train, epochs=10) 

y\_pred=cnn.predict(X\_test)

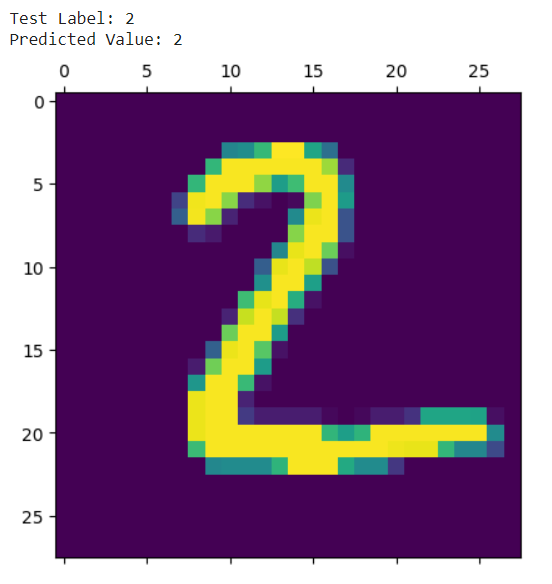
y\_classes=[np.argmax(element) for element in y\_pred]



plt.matshow(X\_test[1])

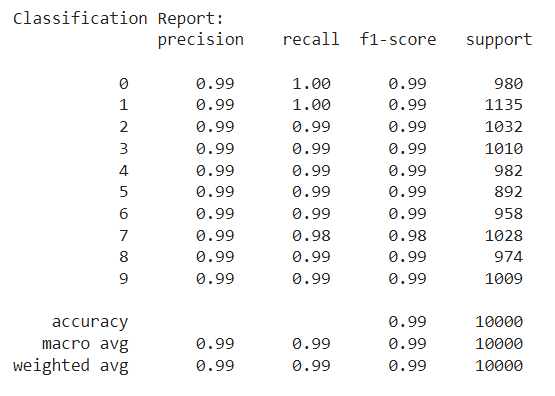
print("Test Label:",y\_test[1])

print("Predicted Value:",y\_classes[1])



from sklearn.metrics import confusion\_matrix, classification\_report

print("Classification Report:\n",classification\_report(y\_test, y\_classes))



1. Image Classification using CIFAR10 dataset

import tensorflow as tf

import keras

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

%matplotlib inline

import numpy as np

(X\_train, y\_train), (X\_test, y\_test)=datasets.cifar10.load\_data()

X\_train.shape, X\_test.shape



classes=["airplane","automobile","bird","cat","deer","dog","frog","horse","ship","truck"]

y\_train=y\_train.reshape(-1,)

y\_train



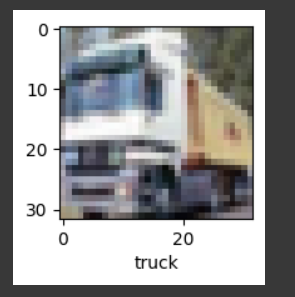
def plot\_sample(X, y,index):

  plt.figure(figsize=(15,2))

  plt.imshow(X[index])

  plt.xlabel(classes[y[index]])

plot\_sample(X\_train, y\_train, 1)



#Normalisation

X\_train=X\_train/255

X\_test=X\_test/255

cnn=models.Sequential([

#cnn

layers.Conv2D(filters=32,kernel\_size=(3,3),activation='relu',input\_shape=(32,32,3)),

layers.MaxPooling2D((2,2)),

layers.Conv2D(filters=64,kernel\_size=(3,3),activation='relu'),

layers.MaxPooling2D((2,2)),

#dense

layers.Flatten(),

layers.Dense(50, activation='relu'),

layers.Dense(10, activation='softmax')

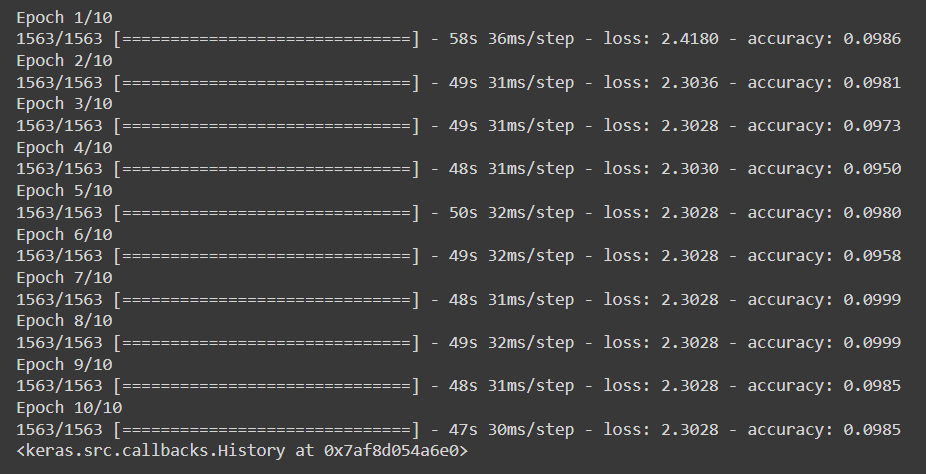
])

cnn.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy',

metrics=['accuracy'])

cnn.fit(X\_train, y\_train, epochs=10)



y\_pred=cnn.predict(X\_test)

y\_classes=[np.argmax(element) for element in y\_pred]

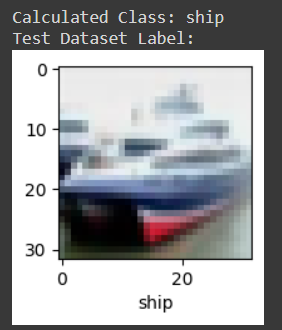


y\_test=y\_test.reshape(-1,)

print("Calculated Class:",classes[y\_classes[1]])

print("Test Dataset Label:")

plot\_sample(X\_test, y\_test, 1)



from sklearn.metrics import confusion\_matrix, classification\_report

print("Classification Report: \n",classification\_report(y\_test, y\_classes))

