1)Develop a Python script to execute various learning rules commonly employed in deep learning, including the Hebbian Learning Rule, Perceptron Learning Rule, Delta Learning Rule, Correlation Learning Rule, and OutStar Learning Rule.

import numpy as np

# Hebbian Learning Rule

def hebbian\_learning\_rule(input\_pattern, weight\_matrix):

return weight\_matrix + np.outer(input\_pattern, input\_pattern)

# Perceptron Learning Rule

def perceptron\_learning\_rule(input\_pattern, target, weight\_vector, learning\_rate):

prediction = np.dot(weight\_vector, input\_pattern)

error = target - prediction

return weight\_vector + learning\_rate \* error \* input\_pattern

# Delta Learning Rule

def delta\_learning\_rule(input\_pattern, target, weight\_matrix, learning\_rate):

prediction = np.dot(weight\_matrix, input\_pattern)

error = target - prediction

return weight\_matrix + learning\_rate \* np.outer(error, input\_pattern)

# Correlation Learning Rule

def correlation\_learning\_rule(input\_pattern, weight\_matrix):

return weight\_matrix + np.outer(input\_pattern, input\_pattern)

# OutStar Learning Rule

def out\_star\_learning\_rule(input\_pattern, weight\_matrix, learning\_rate):

return weight\_matrix + learning\_rate \* np.outer(input\_pattern, input\_pattern)

# Initialize input size

input\_size = 3

# Initialize weights with random values

hebbian\_weights = np.random.rand(input\_size, input\_size)

perceptron\_weights = np.random.rand(input\_size)

delta\_weights = np.random.rand(input\_size, input\_size)

correlation\_weights = np.random.rand(input\_size, input\_size)

out\_star\_weights = np.random.rand(input\_size, input\_size)

# Print initial weights

print("Initial Hebbian Weights:", hebbian\_weights)

print("\nInitial Perceptron Weights:", perceptron\_weights)

print("\nInitial Delta Weights:", delta\_weights)

print("\nInitial Correlation Weights:", correlation\_weights)

print("\nInitial Out Star Weights:", out\_star\_weights)

# Sample input and target data

input\_pattern = np.array([0.2, 0.5, 0.8])

target = 1

# Apply learning rules

hebbian\_weights\_updated = hebbian\_learning\_rule(input\_pattern, hebbian\_weights)

perceptron\_weights\_updated = perceptron\_learning\_rule(input\_pattern, target, perceptron\_weights, learning\_rate=0.1)

delta\_weights\_updated = delta\_learning\_rule(input\_pattern, target, delta\_weights, learning\_rate=0.1)

correlation\_weights\_updated = correlation\_learning\_rule(input\_pattern, correlation\_weights)

out\_star\_weights\_updated = out\_star\_learning\_rule(input\_pattern, out\_star\_weights, learning\_rate=0.1)

# Print updated weights

print("\nUpdated Hebbian Weights:", hebbian\_weights\_updated)

print("\nUpdated Perceptron Weights:", perceptron\_weights\_updated)

print("\nUpdated Delta Weights:", delta\_weights\_updated)

print("\nUpdated Correlation Weights:", correlation\_weights\_updated)

print("\nUpdated Out Star Weights:", out\_star\_weights\_updated)

2)Develop a Python program to implement various activation functions, including the sigmoid, tanh (hyperbolic tangent), ReLU (Rectified Linear Unit), Leaky ReLU, and softmax. The program should include functions to compute the output of each activation function for a given input. Additionally, it should be capable of plotting graphs representing the output of each activation function over a range of input values.

import numpy as np

import matplotlib.pyplot as plt

# Sigmoid function

def sigmoid(x):

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

def plot\_sigmoid():

x = np.linspace(-10, 10, 100) # Generate 100 equally spaced values from -10 to 10

y = sigmoid(x) # Compute the sigmoid function values

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('Sigmoid Output')

plt.title('Sigmoid Activation Function')

plt.grid(True)

plt.show()

# Tanh function

def plot\_tanh():

x = np.linspace(-10, 10, 100)

y = np.tanh(x)

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('Tanh Output')

plt.title('Hyperbolic Tangent (tanh) Activation Function')

plt.grid(True)

plt.show()

# ReLU function

def plot\_relu():

x = np.linspace(-10, 10, 100)

y = np.maximum(0, x)

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('ReLU Output')

plt.title('ReLU Activation Function')

plt.grid(True)

plt.show()

# Leaky ReLU function

def leaky\_relu(x, alpha=0.1):

return np.where(x >= 0, x, alpha \* x)

def plot\_leaky\_relu():

x = np.linspace(-10, 10, 100)

y = leaky\_relu(x)

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('Leaky ReLU Output')

plt.title('Leaky ReLU Activation Function')

plt.grid(True)

plt.show()

# Softmax function

def softmax(x):

e\_x = np.exp(x - np.max(x)) # Subtracting the maximum value for numerical stability

return e\_x / np.sum(e\_x, axis=0)

def plot\_softmax():

x = np.linspace(-2, 2, 100) # Example range for softmax input

y = softmax(x)

plt.plot(x, y)

plt.xlabel('Input')

plt.ylabel('Softmax Output')

plt.title('Softmax Activation Function')

plt.grid(True)

plt.show()

def plot\_softmax\_bar(probabilities, class\_labels):

plt.bar(class\_labels, probabilities)

plt.xlabel("Class")

plt.ylabel("Probability")

plt.title("Softmax Output")

plt.show()

def main\_menu():

while True:

print("\nMAIN MENU")

print("1. Sigmoid")

print("2. Hyperbolic Tangent (tanh)")

print("3. Rectified Linear Unit (ReLU)")

print("4. Leaky ReLU")

print("5. Softmax")

print("6. Exit")

choice = int(input("Enter your choice: "))

if choice == 1:

plot\_sigmoid()

elif choice == 2:

plot\_tanh()

elif choice == 3:

plot\_relu()

elif choice == 4:

plot\_leaky\_relu()

elif choice == 5:

x = np.array([1, 2, 3]) # Example input for softmax

result = softmax(x)

print("Softmax Output:", result)

class\_labels = ["Class A", "Class B", "Class C"]

plot\_softmax\_bar(result, class\_labels)

elif choice == 6:

break

else:

print("Oops! Incorrect Choice.")

# Calling the main menu function

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

main\_menu()

3) Implement a python program for Perceptron Networks by considering the given scenario. A student wants to make a decision about whether to go for a movie or not by looking at 3 parameters using a single neuron. The three inputs are Favorite hero, heroine, and Climate. Each has weights and we have a bias in the perceptron. If the condition is true input is 1 else input is 0, weights for Favorite hero=0.2, heroine=0.4, and Climate=0.2 and bias=-0.5. Output is 1. The decision is to go for a movie.Calculate the Accuracy .

import numpy as np

# Perceptron Network

class Perceptron:

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

self.weights = weights

self.bias = bias

def activation(self, inputs):

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

return 1 if weighted\_sum >= 0 else 0

def train(self, inputs, expected\_outputs, epochs=10, learning\_rate=0.1):

for \_ in range(epochs):

for input\_vector, expected\_output in zip(inputs, expected\_outputs):

predicted\_output = self.activation(input\_vector)

error = expected\_output - predicted\_output

self.weights += learning\_rate \* error \* input\_vector

self.bias += learning\_rate \* error

def predict(self, inputs):

predictions = []

for input\_vector in inputs:

prediction = self.activation(input\_vector)

predictions.append(prediction)

return predictions

# Example usage

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

# Given scenario

weights = np.array([0.2, 0.4, 0.2])

bias = -0.5

# Training data

inputs = np.array([

[1, 1, 1], # Favorite hero, heroine, and good Climate

[1, 0, 1], # Favorite hero, not favorite heroine, good Climate

[0, 1, 0], # Not favorite hero, favorite heroine, bad Climate

[0, 0, 1] # Not favorite hero, not favorite heroine, good Climate

])

expected\_outputs = np.array([1, 0, 0, 0]) # Expected outputs (go for a movie or not)

# Create the Perceptron Network

perceptron = Perceptron(weights, bias)

# Train the Perceptron Network

perceptron.train(inputs, expected\_outputs)

# Test the Perceptron Network

predictions = perceptron.predict(inputs)

# Calculate accuracy

accuracy = sum(predictions == expected\_outputs) / len(expected\_outputs)

print("Accuracy:", accuracy)

4) Write a program in deep learning to apply image processing operations such as Histogram equalization, Thresholding, Edge detection, Data augmentation, Morphological Operations.

import cv2

import numpy as np

# Load the image

image = cv2.imread('download.jpeg')

# Convert to grayscale for operations that need it

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Histogram Equalization for grayscale image

equalized\_gray = cv2.equalizeHist(gray\_image)

cv2.imshow('Histogram Equalization (Gray)', equalized\_gray)

# Thresholding

\_, threshed = cv2.threshold(gray\_image, 127, 255, cv2.THRESH\_BINARY)

cv2.imshow('Thresholding', threshed)

# Edge Detection

edges = cv2.Canny(gray\_image, 100, 200)

cv2.imshow('Edge Detection', edges)

# Data Augmentation

# Rotation

rows, cols = image.shape[:2]

M = cv2.getRotationMatrix2D((cols / 2, rows / 2), 45, 1)

rotated = cv2.warpAffine(image, M, (cols, rows))

cv2.imshow('Rotation', rotated)

# Flipping

flipped = cv2.flip(image, 1)

cv2.imshow('Flipping', flipped)

# Morphological Operations

kernel = np.ones((5, 5), np.uint8)

# Erosion

eroded = cv2.erode(image, kernel, iterations=1)

cv2.imshow('Erosion', eroded)

# Dilation

dilated = cv2.dilate(image, kernel, iterations=1)

cv2.imshow('Dilation', dilated)

# Opening

opening = cv2.morphologyEx(image, cv2.MORPH\_OPEN, kernel)

cv2.imshow('Opening', opening)

# Closing

closing = cv2.morphologyEx(image, cv2.MORPH\_CLOSE, kernel)

cv2.imshow('Closing', closing)

cv2.waitKey(0)

cv2.destroyAllWindows()

5)   
Implement image style transfer, transforming a given content image to adopt the artistic style of another image, using a pre-trained model.

import tensorflow\_hub as hub

import tensorflow as tf

import cv2

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.python.ops.numpy\_ops import np\_config

np\_config.enable\_numpy\_behavior()

# Load the image

def load\_img(path):

img = cv2.imread(path)

img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

img = img / 255.0

return img

content\_image = load\_img('content.jpeg')

style\_image = load\_img('style.jpeg')

# Load the model

model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')

# Apply the style

def apply\_style(content\_image, style\_image):

content\_image = content\_image.reshape(1, content\_image.shape[0], content\_image.shape[1], content\_image.shape[2]).astype('float32')

content\_image = tf.convert\_to\_tensor(content\_image)

style\_image = cv2.resize(style\_image, (256, 256))

style\_image = style\_image.reshape(1, style\_image.shape[0], style\_image.shape[1], style\_image.shape[2]).astype('float32')

style\_image = tf.convert\_to\_tensor(style\_image)

outputs = model(tf.constant(content\_image), tf.constant(style\_image))

stylized\_image = outputs[0]

return stylized\_image

# Display the image

stylized\_img = apply\_style(content\_image, style\_image)

plt.xticks([])

plt.yticks([])

plt.grid(False)

plt.imshow(stylized\_img[0])

plt.show()

6)   
Implement in python SVM/Softmax classifier for CIFAR-10 dataset.

import tensorflow as tf

import numpy as np

from tensorflow.keras.datasets import cifar10

from sklearn.preprocessing import OneHotEncoder

import matplotlib.pyplot as plt

# Load the CIFAR-10 dataset

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

# Define the class names for easier interpretation of predictions

cifar\_10\_classes = [

"Airplane", "Automobile", "Bird", "Cat", "Deer", "Dog", "Frog", "Horse", "Ship", "Truck"

]

# Display the shape of the training data

print(x\_train.shape)

# Display the first image and its class name

plt.imshow(x\_train[0])

plt.title(cifar\_10\_classes[y\_train[0][0]])

plt.axis("off")

plt.show()

# Normalize the pixel values to be between 0 and 1

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

# Convert the class labels to one-hot encoded vectors

one\_hot\_encoder = OneHotEncoder()

y\_train = one\_hot\_encoder.fit\_transform(y\_train).toarray()

y\_test = one\_hot\_encoder.transform(y\_test).toarray()

# Build a simple neural network model with a single softmax layer

softmax\_model = tf.keras.models.Sequential([

tf.keras.layers.Flatten(input\_shape=(32, 32, 3)),

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

])

# Compile the model

softmax\_model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Train the model for 20 epochs with a batch size of 64, using a validation split for evaluation

softmax\_model.fit(x\_train, y\_train, epochs=20, batch\_size=64, validation\_data=(x\_test, y\_test))

# Making a prediction using the trained model

new\_image = x\_test[10]

plt.imshow(new\_image)

plt.axis("off")

plt.show()

img = np.expand\_dims(new\_image, axis=0)

pred = softmax\_model.predict(img)

prediction = np.argmax(pred)

print(f"Predicted class: {cifar\_10\_classes[prediction]}")

7) Develop a convolutional neural network (CNN) model to classify handwritten digits using the MNIST dataset. The goal is to train a model that accurately identifies digits (0-9) from images.

import pandas as pd

import tensorflow as tf

import matplotlib.pyplot as plt

import tensorflow.keras as keras

import numpy as np

# Load the dataset and divide into train and test

dataset = keras.datasets.mnist

class\_names = ['Zero', 'One', 'Two', 'Three', 'Four', 'Five', 'Six', 'Seven', 'Eight', 'Nine']

(x\_train, y\_train), (x\_test, y\_test) = dataset.load\_data()

# Reshape the data to add a channel dimension

x\_train = x\_train.reshape((x\_train.shape[0], x\_train.shape[1], x\_train.shape[2], 1))

x\_test = x\_test.reshape((x\_test.shape[0], x\_test.shape[1], x\_test.shape[2], 1))

# Print the shape of the training and testing data

print(f"Training data shape: {x\_train.shape}")

print(f"Testing data shape: {x\_test.shape}")

# Plot five data samples with their class names

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

for i in range(9):

plt.subplot(3, 3, i + 1)

plt.imshow(x\_train[i].reshape(28, 28), cmap='gray')

plt.title(class\_names[y\_train[i]])

plt.axis("off")

plt.show()

# Normalize the pixel values to be between 0 and 1

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

# Define the model

model = keras.models.Sequential([

keras.layers.Conv2D(64, (3, 3), input\_shape=(28, 28, 1), activation="relu"),

keras.layers.MaxPooling2D(pool\_size=(2, 2), strides=2),

keras.layers.Conv2D(64, (3, 3), activation="relu"),

keras.layers.MaxPooling2D(pool\_size=(2, 2), strides=2),

keras.layers.Flatten(),

keras.layers.Dense(64, activation="relu"),

keras.layers.Dense(10, activation="softmax")

])

# Compile the model

model.compile(optimizer="adam",

loss=tf.keras.losses.SparseCategoricalCrossentropy(),

metrics=["accuracy"])

# Train the model

model.fit(x\_train, y\_train, epochs=5, callbacks=[keras.callbacks.EarlyStopping(patience=2)])

# Evaluate the model

loss, accuracy = model.evaluate(x\_test, y\_test)

print(f"Test loss: {loss}")

print(f"Test accuracy: {accuracy}")

# Make a prediction on a sample image from the test set

sample\_img = x\_test[0]

plt.imshow(sample\_img.reshape(28, 28), cmap='gray')

plt.title("Sample Image")

plt.axis("off")

plt.show()

# Prepare the image for prediction

img = np.expand\_dims(sample\_img, axis=0)

# Make prediction

pred = model.predict(img)

predicted\_class = np.argmax(pred)

print(f"Predicted: {class\_names[predicted\_class]}\nActual: {class\_names[y\_test[0]]}")

# Print model summary

model.summary()

8)

import numpy as np

import pandas as pd

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout

# Load the dataset

data = pd.read\_csv('sonar\_dataset.csv', header=None)

X = data.iloc[:, :-1].values

y = data.iloc[:, -1].values

# Encode the target variable

encoder = LabelEncoder()

y = encoder.fit\_transform(y)

# Split the data into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Define the model architecture

model = Sequential([

Dense(64, activation='relu', input\_shape=(X\_train.shape[1],)),

Dropout(0.2),

Dense(32, activation='relu'),

Dropout(0.2),

Dense(1, activation='sigmoid')

])

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=100, batch\_size=32, validation\_data=(X\_test, y\_test))

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test, verbose=0)

print('Test accuracy:', test\_acc)

10)

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow.keras.optimizers import Adam

# Load MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Preprocess data

x\_train = x\_train.reshape(-1, 28, 28, 1) / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1) / 255.0

y\_train = keras.utils.to\_categorical(y\_train)

y\_test = keras.utils.to\_categorical(y\_test)

# Model without dropout

model\_without\_dropout = Sequential([

Flatten(input\_shape=(28, 28, 1)),

Dense(128, activation='relu'),

Dense(64, activation='relu'),

Dense(10, activation='softmax')

])

# Model with dropout

model\_with\_dropout = Sequential([

Flatten(input\_shape=(28, 28, 1)),

Dense(128, activation='relu'),

Dropout(0.2),

Dense(64, activation='relu'),

Dropout(0.2),

Dense(10, activation='softmax')

])

# Compile and train models

optimizer = Adam()

model\_without\_dropout.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])

model\_with\_dropout.compile(optimizer=optimizer, loss='categorical\_crossentropy', metrics=['accuracy'])

history\_without\_dropout = model\_without\_dropout.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))

history\_with\_dropout = model\_with\_dropout.fit(x\_train, y\_train, epochs=10, validation\_data=(x\_test, y\_test))