**PRACTICAL JOURNAL**

in

**Deep Learning**

Submitted to

**Laxman Devram Sonawane College, Kalyan (W) 421301**

in partial fulfilment for the award of the degree of

** Master of Science in Information Technology**

(Affiliated to Mumbai University)

*Submitted by*

**TANIA TAZIM JAMDAR**

Under the guidance of

**Dr. Priyanka Pawar**

Department of Information Technology

Kalyan, Maharashtra

Academic Year 2024-25



The Kalyan Wholesale Merchants Education Society’s

**Laxman Devram Sonawane College,**

**Kalyan (W) 421301**

**Department of Information Technology**

**Masters of Science – Part II**

**Certificate**

This is to certify that **Mr. Tania Tazim Jamdar**, Seat number**\_\_\_\_\_\_\_\_\_\_\_\_\_**, studying in Masters of Science in Information Technology Part II, Semester IV has satisfactorily completed the practical of “**Deep Learning** ” as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge Coordinator In-charge ExternalExaminer

College Seal

**INDEX**

|  |  |  |
| --- | --- | --- |
| SR No. | Practical List | Sign |
| 1 | **Introduction to TensorFlow** |  |
| a. | * Create tensors with different shapes and data types. * Perform basic operations like addition, subtraction, multiplication, and division on tensors. * Reshape, slice, and index tensors to extract specific elements or sections * Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow |  |
| b. | Program to solve the XOR problem |  |
| 2 | **Linear Regression** |  |
| a. | * Implement a simple linear regression model using TensorFlow's lowlevel API (or tf. keras). * Train the model on a toy dataset (e.g., housing prices vs. square footage) * Visualize the loss function and the learned linear relationship. |  |
| b. | Make predictions on new data points |  |
| 3 | **Convolutional Neural Networks (Classification)** |  |
| a. | Implementing deep neural network for performing binary classification task |  |
| b. | Using a deep feed-forward network with two hidden layers for performing multiclass classification and predicting the class. |  |
| 4 | Write a program to implement deep learning Techniques for image segmentation. O |  |
| 5 | Write a program to predict a caption for a sample image using LSTM |  |
| 6 | Applying the Autoencoder algorithms for encoding real-world data |  |
| 7 | Write a program for character recognition using RNN and compare it with CNN. |  |
| 8 | Write a program to develop Autoencoders using MNIST Handwritten Digits |  |
| 9 | Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.(google stock price) |  |
| 10 | Applying Generative Adversarial Networks for image generation and unsupervised tasks. |  |

**Practical 1 : Introduction to TensorFlow**

**1-a1. Create tensors with different shapes and data types**

**Code :**

import tensorflow as tf

**# Create a scalar (0-D tensor)**

scalar = tf.constant(42)

print("Scalar (0-D Tensor):", scalar)

**# Create a vector (1-D tensor)**

vector = tf.constant([1.5, 2.5, 3.5], dtype=tf.float32)

print("Vector (1-D Tensor):", vector)

**# Create a matrix (2-D tensor)**

matrix = tf.constant([[1, 2], [3, 4], [5, 6]], dtype=tf.int32)

print("Matrix (2-D Tensor):", matrix)

**# Create a 3-D tensor**

tensor\_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print("3-D Tensor:", tensor\_3d)

**# Create a tensor of all zeros**

zeros\_tensor = tf.zeros([2, 3])

print("Zeros Tensor:", zeros\_tensor)

**# Create a tensor of all ones**

ones\_tensor = tf.ones([3, 2, 2])

print("Ones Tensor:", ones\_tensor)

**# Create a tensor with random values**

random\_tensor = tf.random.normal([2, 2], mean=0, stddev=1)

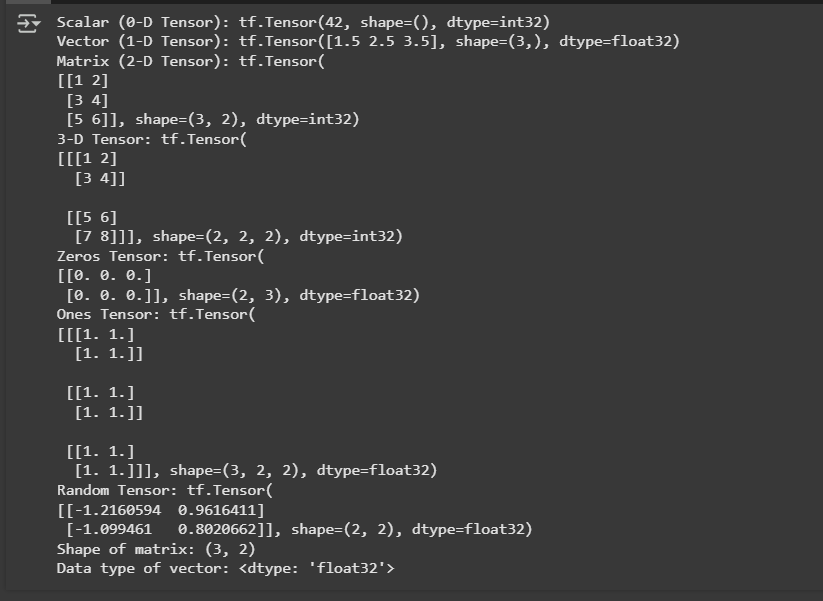
print("Random Tensor:", random\_tensor)

**# Get the shape and data type of a tensor**

print("Shape of matrix:", matrix.shape)

print("Data type of vector:", vector.dtype)

**Output :**

****

**1-a2. Perform basic operations like addition, subtraction, multiplication, and division on tensors.**

**Code :**

import tensorflow as tf

**# Define two tensors**

a = tf.constant([3, 6, 9], dtype=tf.int32)

b = tf.constant([2, 4, 6], dtype=tf.int32)

**# Perform basic arithmetic operations**

addition = tf.add(a, b)

subtraction = tf.subtract(a, b)

multiplication = tf.multiply(a, b)

division = tf.divide(a, b)

**# Display the results**

print("Tensor A:", a.numpy())

print("Tensor B:", b.numpy())

print("Addition:", addition.numpy())

print("Subtraction:", subtraction.numpy())

print("Multiplication:", multiplication.numpy())

print("Division:", division.numpy())

**# More tensor operations**

squared = tf.square(a)

sqrt\_b = tf.sqrt(tf.cast(b, tf.float32))

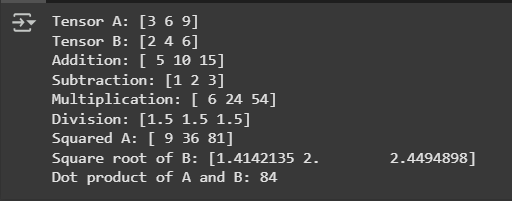
dot\_product = tf.tensordot(a, b, axes=1)

print("Squared A:", squared.numpy())

print("Square root of B:", sqrt\_b.numpy())

print("Dot product of A and B:", dot\_product.numpy())

**Output :**

****

**1-a3. Reshape, slice, and index tensors to extract specific elements or sections**

**Code :**

import tensorflow as tf

**# Create a sample tensor**

original\_tensor = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Original Tensor:\n", original\_tensor.numpy())

**# Reshape the tensor**

reshaped\_tensor = tf.reshape(original\_tensor, (1, 9))

print("\nReshaped Tensor (1x9):\n", reshaped\_tensor.numpy())

**# Slice the tensor (Extract rows 1 and 2, columns 1 and 2)**

sliced\_tensor = original\_tensor[1:3, 1:3]

print("\nSliced Tensor (Rows 1-2, Columns 1-2):\n", sliced\_tensor.numpy())

**# Indexing (Extract specific elements)**

first\_element = original\_tensor[0, 0]

last\_row = original\_tensor[-1]

print("\nFirst Element (0,0):", first\_element.numpy())

print("Last Row:", last\_row.numpy())

**# Use tf.gather to extract specific indices**

gathered\_elements = tf.gather(original\_tensor, [0, 2], axis=0)

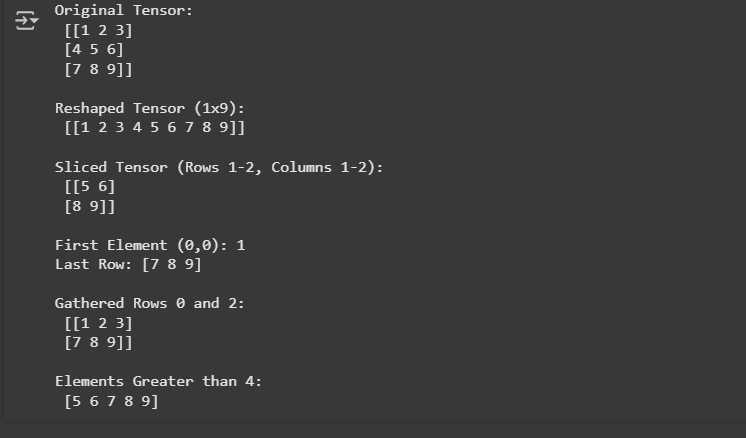
print("\nGathered Rows 0 and 2:\n", gathered\_elements.numpy())

**# Use tf.boolean\_mask to extract elements with a condition**

masked\_elements = tf.boolean\_mask(original\_tensor, original\_tensor > 4)

print("\nElements Greater than 4:\n", masked\_elements.numpy())

**Output :**

****

**1-a4. Performing matrix multiplication and finding eigenvectors and eigenvalues using TensorFlow**

**Code :**

import tensorflow as tf

import numpy as np

**# Define two matrices**

matrix\_a = tf.constant([[3, 4], [5, 6]], dtype=tf.float32)

matrix\_b = tf.constant([[7, 8], [9, 10]], dtype=tf.float32)

**# Perform matrix multiplication**

matrix\_product = tf.matmul(matrix\_a, matrix\_b)

**# Display the result of matrix multiplication**

print("Matrix A:", matrix\_a.numpy())

print("Matrix B:", matrix\_b.numpy())

print("Matrix Product (A \* B):", matrix\_product.numpy())

**# Finding eigenvalues and eigenvectors**

matrix\_c = tf.constant([[4, -2], [1, 1]], dtype=tf.float32)

**# Convert TensorFlow tensor to NumPy array for eigenvalue computation**

matrix\_c\_np = matrix\_c.numpy()

eigenvalues, eigenvectors = np.linalg.eig(matrix\_c\_np)

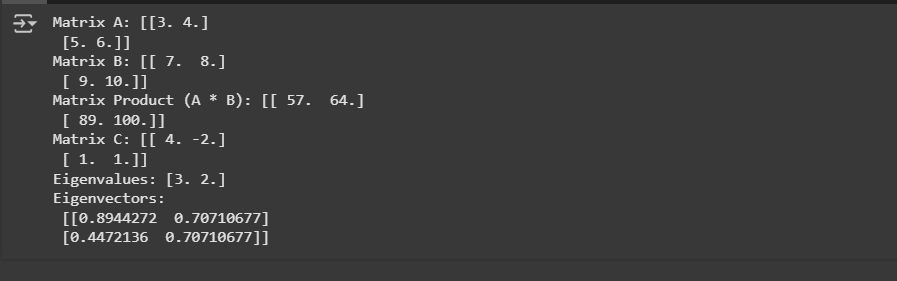
**# Display eigenvalues and eigenvectors**

print("Matrix C:", matrix\_c\_np)

print("Eigenvalues:", eigenvalues)

print("Eigenvectors:\n", eigenvectors)

**Output :**

****

**1b. Program to solve the XOR problem**

**Code :**

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

import numpy as np

**# Define XOR inputs and outputs**

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)

y = np.array([[0], [1], [1], [0]], dtype=np.float32)

**# Define a simple neural network model**

model = keras.Sequential([

    layers.Dense(4, activation='relu', input\_shape=(2,)),

    layers.Dense(4, activation='relu'),

    layers.Dense(1, activation='sigmoid')])

**# Compile the model**

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

**# Train the model**

history = model.fit(X, y, epochs=1000, verbose=0)

**# Evaluate the model**

loss, accuracy = model.evaluate(X, y)

print(f"Final Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

**# Make predictions**

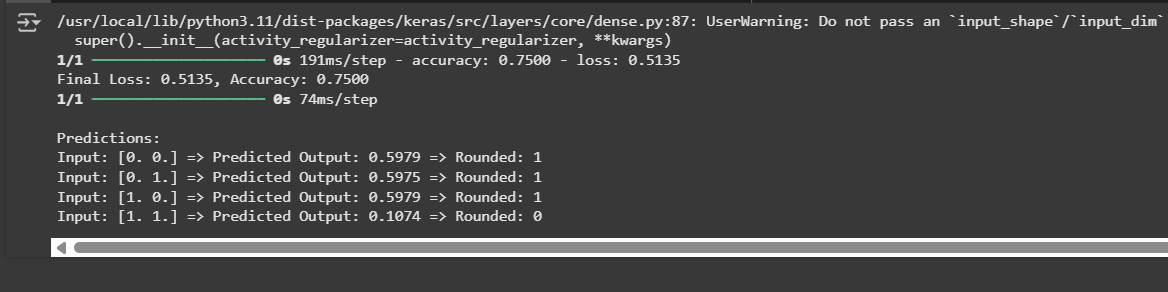
predictions = model.predict(X)

print("\nPredictions:")

for i, p in enumerate(predictions):

    print(f"Input: {X[i]} => Predicted Output: {p[0]:.4f} => Rounded: {int(np.round(p[0]))}")

**Output :**

****

**Practical 2 : Linear Regression**

**2-a1. Implement a simple linear regression model using TensorFlow's lowlevel API (or tf. keras).**

**Code :**

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

**# Generate synthetic data**

np.random.seed(42)

X = np.random.rand(100, 1).astype(np.float32)

y = 3 \* X + 2 + np.random.normal(0, 0.1, (100, 1)).astype(np.float32)

**# Define a simple linear regression model**

model = tf.keras.Sequential([

    tf.keras.layers.Dense(1, input\_shape=(1,))])

**# Compile the model**

model.compile(optimizer='sgd', loss='mse')

**# Train the model**

history = model.fit(X, y, epochs=200, verbose=0)

**# Get the model's weights**

W, b = model.layers[0].get\_weights()

print(f"Learned Weight: {W[0][0]:.2f}, Learned Bias: {b[0]:.2f}")

**# Make predictions**

y\_pred = model.predict(X)

**# Plot the results**

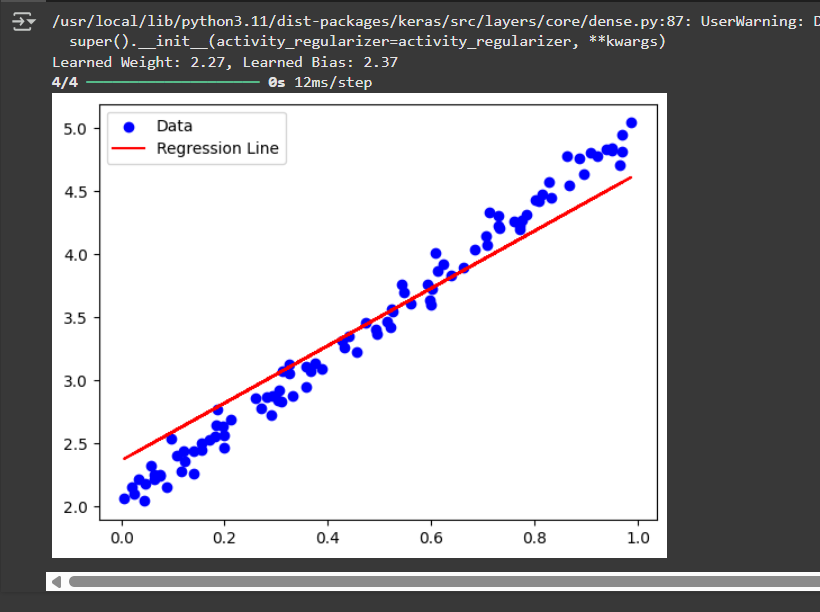
plt.scatter(X, y, color='blue', label='Data')

plt.plot(X, y\_pred, color='red', label='Regression Line')

plt.legend()

plt.show()

**Output :**

****

**2-a2 Train the model on a toy dataset (e.g., housing prices vs. square footage).**

**Code :**

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

**# Toy dataset: square footage vs. housing prices**

square\_feet = np.array([600, 800, 1000, 1200, 1500, 1800, 2000, 2200, 2500], dtype=np.float32)

prices = np.array([150000, 200000, 250000, 300000, 350000, 400000, 450000, 475000, 500000], dtype=np.float32)

**# Reshape data**

X = square\_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)

**# Define a simple linear regression model**

model = tf.keras.Sequential([

    tf.keras.layers.Dense(1, input\_shape=(1,))])

**# Compile the model**

model.compile(optimizer='adam', loss='mse')

**# Train the model**

history = model.fit(X, y, epochs=500, verbose=0)

**# Get the model's weights**

W, b = model.layers[0].get\_weights()

print(f"Learned Weight: {W[0][0]:.2f}, Learned Bias: {b[0]:.2f}")

**# Make predictions**

y\_pred = model.predict(X)

**# Plot the results**

plt.scatter(X, y, color='blue', label='Data')

plt.plot(X, y\_pred, color='red', label='Regression Line')

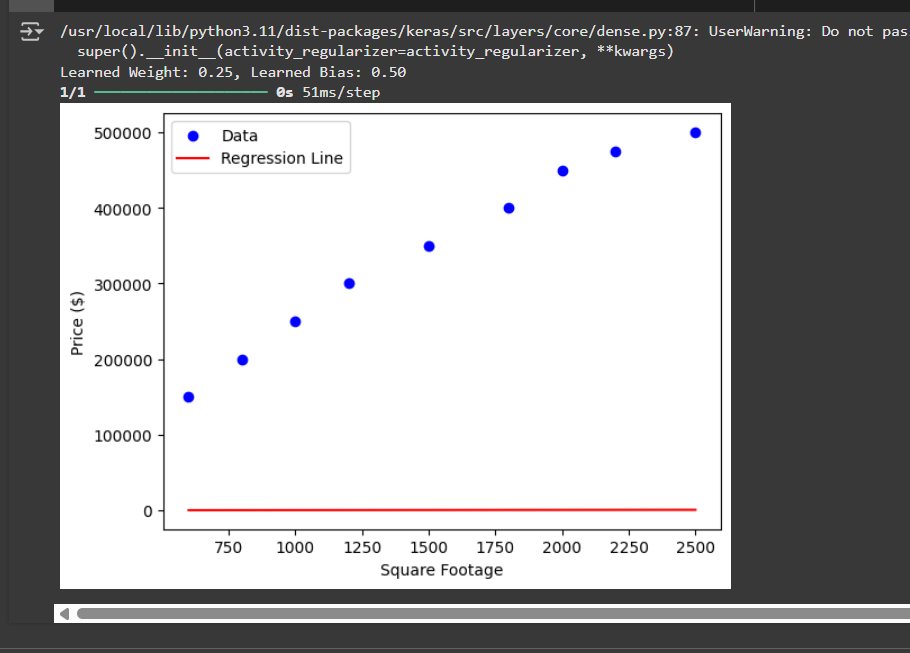
plt.xlabel('Square Footage')

plt.ylabel('Price ($)')

plt.legend()

plt.show()

**Output :**

****

**2-a3. Visualize the loss function and the learned linear relationship.**

**Code :**

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

**# Toy dataset: square footage vs. housing prices**

square\_feet = np.array([600, 800, 1000, 1200, 1500, 1800, 2000, 2200, 2500], dtype=np.float32)

prices = np.array([150000, 200000, 250000, 300000, 350000, 400000, 450000, 475000, 500000], dtype=np.float32)

**# Reshape data**

X = square\_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)

**# Define a simple linear regression model**

model = tf.keras.Sequential([

    tf.keras.layers.Dense(1, input\_shape=(1,))])

**# Compile the model**

model.compile(optimizer='adam', loss='mse')

**# Train the model**

history = model.fit(X, y, epochs=500, verbose=0)

**# Get the model's weights**

W, b = model.layers[0].get\_weights()

print(f"Learned Weight: {W[0][0]:.2f}, Learned Bias: {b[0]:.2f}")

**# Make predictions**

y\_pred = model.predict(X)

**# Plot the results**

plt.figure(figsize=(12, 5))

**# Plot the data and regression line**

plt.subplot(1, 2, 1)

plt.scatter(X, y, color='blue', label='Data')

plt.plot(X, y\_pred, color='red', label='Regression Line')

plt.xlabel('Square Footage')

plt.ylabel('Price ($)')

plt.legend()

plt.title('Linear Regression Fit')

**# Plot the loss function**

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], color='orange')

plt.xlabel('Epochs')

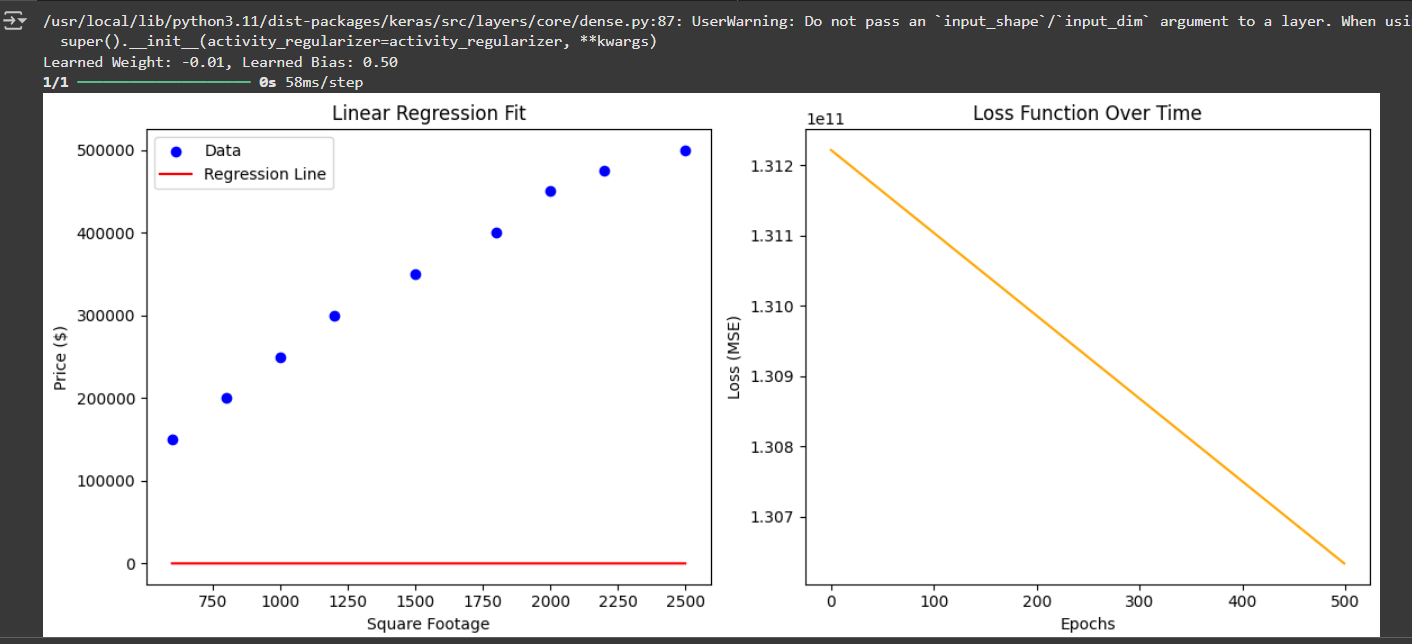
plt.ylabel('Loss (MSE)')

plt.title('Loss Function Over Time')

plt.tight\_layout()

plt.show()

**Output :**

****

**2-a4. Make predictions on new data points**

**Code :**

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

**# Toy dataset: square footage vs. housing prices**

square\_feet = np.array([600, 800, 1000, 1200, 1500, 1800, 2000, 2200, 2500], dtype=np.float32)

prices = np.array([150000, 200000, 250000, 300000, 350000, 400000, 450000, 475000, 500000], dtype=np.float32)

**# Reshape data**

X = square\_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)

**# Define a simple linear regression model**

model = tf.keras.Sequential([

    tf.keras.layers.Dense(1, input\_shape=(1,))])

**# Compile the model**

model.compile(optimizer='adam', loss='mse')

**# Train the model**

history = model.fit(X, y, epochs=500, verbose=0)

**# Get the model's weights**

W, b = model.layers[0].get\_weights()

print(f"Learned Weight: {W[0][0]:.2f}, Learned Bias: {b[0]:.2f}")

**# Make predictions**

y\_pred = model.predict(X)

**# Visualize the results**

plt.figure(figsize=(12, 5))

**# Plot the data and regression line**

plt.subplot(1, 2, 1)

plt.scatter(X, y, color='blue', label='Data')

plt.plot(X, y\_pred, color='red', label='Regression Line')

plt.xlabel('Square Footage')

plt.ylabel('Price ($)')

plt.legend()

plt.title('Linear Regression Fit')

**# Plot the loss function**

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], color='orange')

plt.xlabel('Epochs')

plt.ylabel('Loss (MSE)')

plt.title('Loss Function Over Time')

plt.tight\_layout()

plt.show()

**# Make predictions on new data points**

new\_data = np.array([[1600], [2100], [3000]], dtype=np.float32)

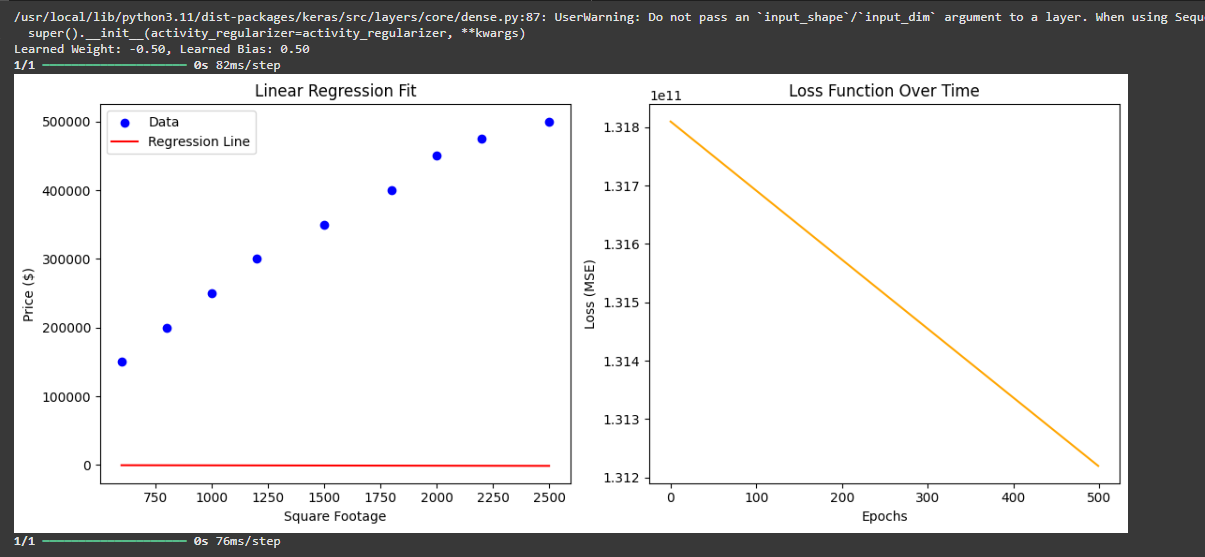
new\_predictions = model.predict(new\_data)

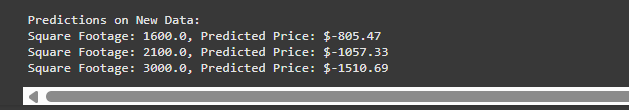
print("\nPredictions on New Data:")

for sqft, price in zip(new\_data.flatten(), new\_predictions.flatten()):

    print(f"Square Footage: {sqft}, Predicted Price: ${price:.2f}")

**Output :**

****

****

**Practical 3 : Convolutional Neural Networks (Classification)**

**3a.** **Implementing deep neural network for performing binary classification task**

**Code :**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import cifar10

import matplotlib.pyplot as plt

**# Load and preprocess data**

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

**# For binary classification, let's classify 'airplane' (class 0) vs 'automobile' (class 1)**

**# Reshape y\_trrain & y\_test to 1D arrays**

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

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

X\_train = X\_train[(y\_train == 0) | (y\_train == 1)].astype('float32') / 255.0

X\_test = X\_test[(y\_test == 0) | (y\_test == 1)].astype('float32') / 255.0

y\_train = y\_train[(y\_train == 0) | (y\_train == 1)]

y\_test = y\_test[(y\_test == 0) | (y\_test == 1)]

**# Build CNN model**

model = models.Sequential([

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

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

    layers.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=10, batch\_size=32, validation\_data=(X\_test, y\_test), verbose=2)

**# Evaluate the model**

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

print(f"Test Accuracy: {accuracy:.2f}")

**# Plot training history**

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

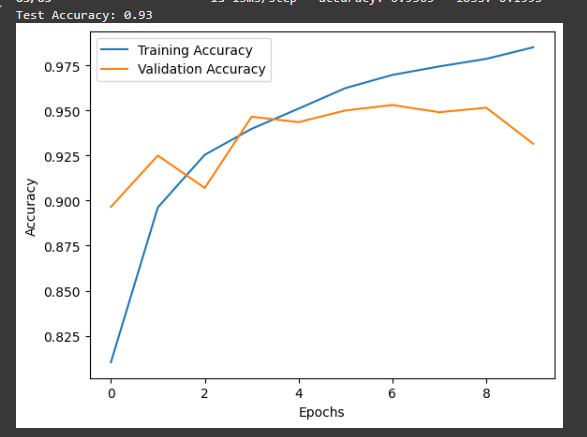
plt.ylabel('Accuracy')

plt.legend()

plt.show()

**Output :**

****

****

**3b. Using a deep feed-forward network with two hidden layers for performing multiclass classification and predicting the class.**

**Code :**

import tensorflow as tf

from tensorflow.keras import layers, models

from tensorflow.keras.datasets import mnist

import matplotlib.pyplot as plt

**# Load and preprocess data**

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

**# Normalize the data**

X\_train = X\_train.reshape(-1, 28 \* 28).astype('float32') / 255.0

X\_test = X\_test.reshape(-1, 28 \* 28).astype('float32') / 255.0

**# Define a deep feed-forward network model**

model = models.Sequential([

    layers.Dense(128, activation='relu', input\_shape=(28 \* 28,)),

    layers.Dense(64, activation='relu'),

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

**# Compile the model**

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

**# Train the model**

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(X\_test, y\_test), verbose=2)

**# Evaluate the model**

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

print(f"Test Accuracy: {accuracy:.2f}")

**# Make predictions on new data**

predictions = model.predict(X\_test[:5])

print("Predicted classes:", [tf.argmax(p).numpy() for p in predictions])

**# Plot training history**

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

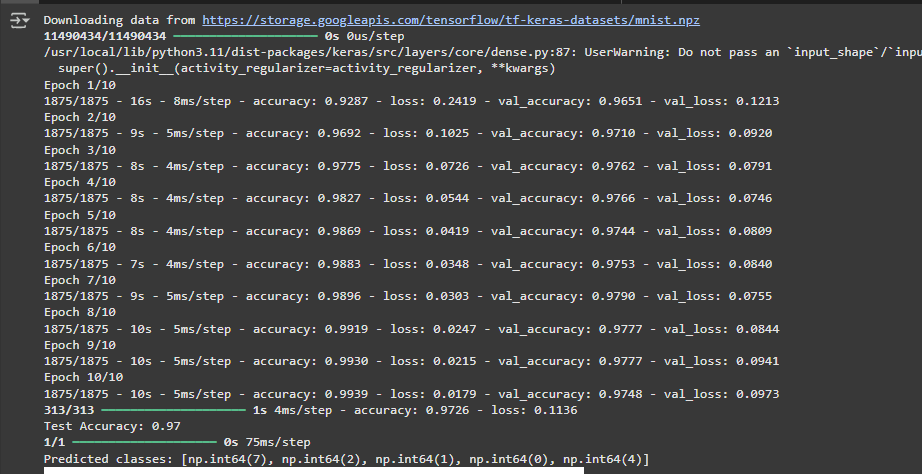
plt.xlabel('Epochs')

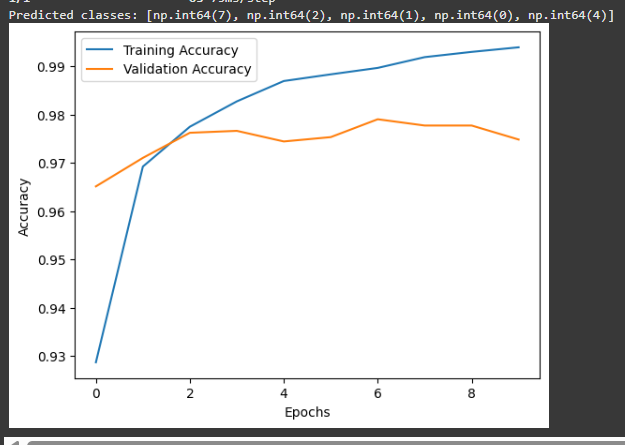
plt.ylabel('Accuracy')

plt.legend()

plt.show()

**Output :**

****

****

**Practical 4**

**Write a program to implement deep learning Techniques for image segmentation.**

**Code :**

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate

import numpy as np

import matplotlib.pyplot as plt

**# Generate a small synthetic dataset**

X\_train = np.random.rand(10, 128, 128, 3)

y\_train = np.random.randint(0, 2, (10, 128, 128, 1))

X\_test = np.random.rand(2, 128, 128, 3)

y\_test = np.random.randint(0, 2, (2, 128, 128, 1))

**# Define a simple U-Net model**

def build\_simple\_unet():

    inputs = Input(shape=(128, 128, 3))

    c1 = Conv2D(16, (3, 3), activation='relu', padding='same')(inputs)

    p1 = MaxPooling2D((2, 2))(c1)

    c2 = Conv2D(32, (3, 3), activation='relu', padding='same')(p1)

    p2 = MaxPooling2D((2, 2))(c2)

    c3 = Conv2D(64, (3, 3), activation='relu', padding='same')(p2)

    u1 = UpSampling2D((2, 2))(c3)

    u1 = concatenate([u1, c2])

    c4 = Conv2D(32, (3, 3), activation='relu', padding='same')(u1)

    u2 = UpSampling2D((2, 2))(c4)

    u2 = concatenate([u2, c1])

    c5 = Conv2D(16, (3, 3), activation='relu', padding='same')(u2)

    outputs = Conv2D(1, (1, 1), activation='sigmoid')(c5)

    model = Model(inputs, outputs)

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

    return model

model = build\_simple\_unet()

model.summary()

**# Train the model**

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=2, validation\_data=(X\_test, y\_test))

**# Test on a new image**

def visualize\_segmentation(model, image):

    pred\_mask = model.predict(np.expand\_dims(image, axis=0))[0]

    pred\_mask = (pred\_mask > 0.5).astype(np.uint8)

    plt.figure(figsize=(10, 5))

    plt.subplot(1, 2, 1)

    plt.title('Original Image')

    plt.imshow(image)

    plt.subplot(1, 2, 2)

    plt.title('Predicted Mask')

    plt.imshow(pred\_mask.squeeze(), cmap='gray')

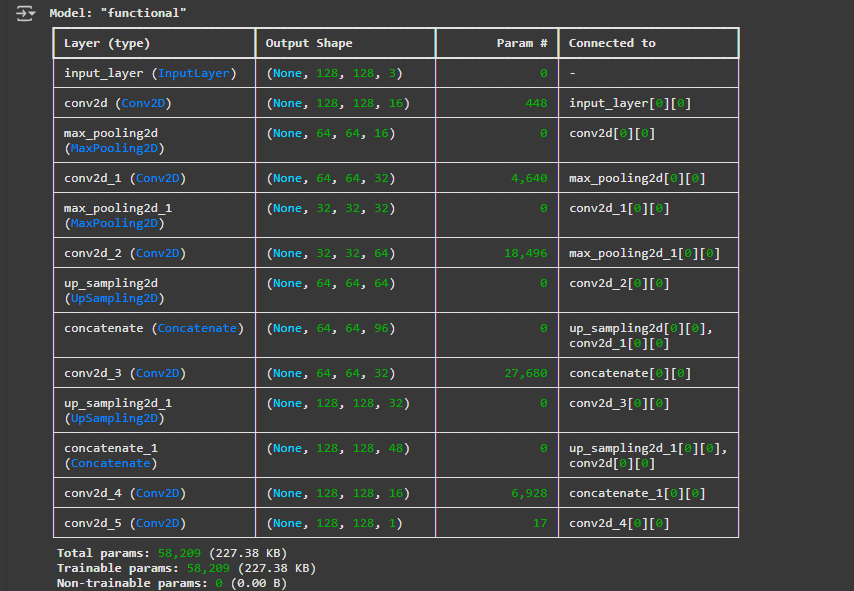
    plt.show()

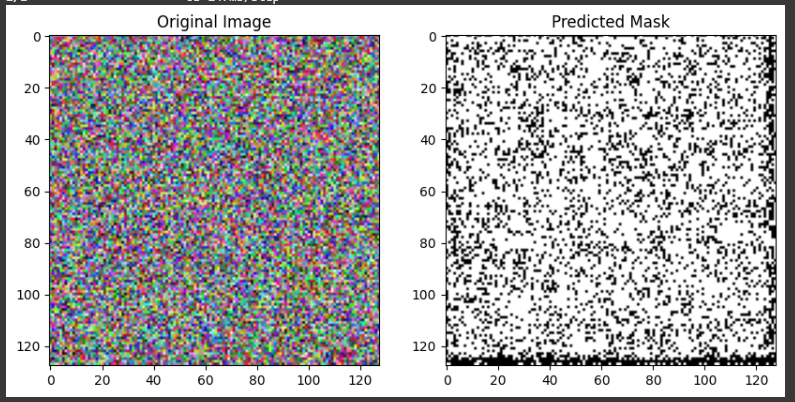
**# Test on a random image**

random\_idx = np.random.randint(0, X\_test.shape[0])

visualize\_segmentation(model, X\_test[random\_idx])

**Output :**

****

****

**Practical 5**

**Write a program to predict a caption for a sample image using LSTM.**

**Code :**

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, LSTM, Embedding, Dropout, add

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.preprocessing.image import load\_img, img\_to\_array

from tensorflow.keras.applications.vgg16 import VGG16, preprocess\_input

import tensorflow as tf

**# Define a small dataset**

images = ["/content/Dog.jpg", "/content/cat.jpg", "/content/bike.jpg"]

captions = [

    "startseq a dog running in the park endseq",

    "startseq a cat sitting on a couch endseq",

    "startseq a person riding a bike endseq"

]

**# Load VGG16 model for image feature extraction**

base\_model = VGG16(weights='imagenet')

model\_vgg = Model(inputs=base\_model.input, outputs=base\_model.layers[-2].output)

def extract\_features(image\_path):

    img = load\_img(image\_path, target\_size=(224, 224))

    img\_array = img\_to\_array(img)

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

    img\_array = preprocess\_input(img\_array)

    return model\_vgg.predict(img\_array)

**# Tokenize the captions**

tokenizer = Tokenizer()

tokenizer.fit\_on\_texts(captions)

max\_length = max(len(c.split()) for c in captions)

vocab\_size = len(tokenizer.word\_index) + 1

**# Convert captions to sequences**

sequences = tokenizer.texts\_to\_sequences(captions)

padded\_captions = pad\_sequences(sequences, maxlen=max\_length, padding='post')

**# Define the model**

embedding\_dim = 256

hidden\_units = 256

image\_input = Input(shape=(4096,))

image\_output = Dense(embedding\_dim, activation='relu')(image\_input)

caption\_input = Input(shape=(max\_length,))

embedding = Embedding(vocab\_size, embedding\_dim, mask\_zero=True)(caption\_input)

lstm\_out = LSTM(hidden\_units)(embedding)

combined = add([image\_output, lstm\_out])

decoder\_output = Dense(vocab\_size, activation='softmax')(combined)

model = Model(inputs=[image\_input, caption\_input], outputs=decoder\_output)

model.compile(loss='categorical\_crossentropy', optimizer='adam')

**# Dummy training labels**

y\_train = np.zeros((3, vocab\_size))

model.fit([np.random.rand(3, 4096), padded\_captions], y\_train, epochs=5, batch\_size=1)

**# Generate a caption**

def generate\_caption(model, tokenizer, image\_feature, max\_length):

    in\_text = 'startseq'

    for \_ in range(max\_length):

        sequence = tokenizer.texts\_to\_sequences([in\_text])[0]

        sequence = pad\_sequences([sequence], maxlen=max\_length)

        yhat = np.argmax(model.predict([image\_feature, sequence], verbose=0))

        word = tokenizer.index\_word.get(yhat, None)

        if word is None:

            break

        in\_text += ' ' + word

        if word == 'endseq':

            break

    return in\_text.replace('startseq', '').replace('endseq', '').strip()

**# Test on a sample image**

sample\_image\_path = images[0]

plt.imshow(load\_img(sample\_image\_path))

sample\_feature = extract\_features(sample\_image\_path)

caption = generate\_caption(model, tokenizer, sample\_feature, max\_length)

print("Predicted Caption:", caption)

**Output :**

****

**Practical 6**

**Applying the Autoencoder algorithms for encoding real-world data**

**Code :**

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Flatten, Reshape

from tensorflow.keras.datasets import mnist

**# Load and preprocess the MNIST dataset**

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

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))

x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))

**# Define the Autoencoder architecture**

input\_img = Input(shape=(28, 28, 1))

**# Encoder**

x = Flatten()(input\_img)

x = Dense(128, activation='relu')(x)

x = Dense(64, activation='relu')(x)

encoded = Dense(32, activation='relu')(x)

**# Decoder**

x = Dense(64, activation='relu')(encoded)

x = Dense(128, activation='relu')(x)

x = Dense(28 \* 28, activation='sigmoid')(x)

decoded = Reshape((28, 28, 1))(x)

**# Define the Autoencoder model**

autoencoder = Model(input\_img, decoded)

**# Compile the model**

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

**# Train the Autoencoder**

autoencoder.fit(

    x\_train, x\_train,

    epochs=50,

    batch\_size=256,

    shuffle=True,

    validation\_data=(x\_test, x\_test)

)

**# Encode and decode some images**

encoded\_imgs = autoencoder.predict(x\_test)

**# Display original and reconstructed images**

n = 10

plt.figure(figsize=(20, 4))

for i in range(n):

    # Original

    ax = plt.subplot(2, n, i + 1)

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

    plt.axis('off')

**# Reconstructed**

    ax = plt.subplot(2, n, i + 1 + n)

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

    plt.axis('off')

plt.show()

**Output :**

****

**Practical 7**

**Write a program for character recognition using RNN and compare it with CNN.**

**Code :**

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, LSTM, Conv2D, MaxPooling2D, Flatten, Reshape, Dropout

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

**# Load and preprocess the MNIST dataset**

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

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

**# Reshape data for CNN**

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

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

**# Reshape data for RNN**

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

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

**# One-hot encode the labels**

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

**# Define RNN model**

rnn\_model = Sequential([

    LSTM(128, input\_shape=(28, 28)),

    Dense(64, activation='relu'),

    Dense(10, activation='softmax')])

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

**# Define CNN model**

cnn\_model = Sequential([

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

    MaxPooling2D(pool\_size=(2, 2)),

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

    MaxPooling2D(pool\_size=(2, 2)),

    Flatten(),

    Dense(128, activation='relu'),

    Dropout(0.5),

    Dense(10, activation='softmax')])

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

**# Train the RNN model**

print("Training RNN model...")

rnn\_history = rnn\_model.fit(x\_train\_rnn, y\_train, epochs=10, batch\_size=128, validation\_data=(x\_test\_rnn, y\_test))

**# Train the CNN model**

print("Training CNN model...")

cnn\_history = cnn\_model.fit(x\_train\_cnn, y\_train, epochs=10, batch\_size=128, validation\_data=(x\_test\_cnn, y\_test))

**# Evaluate the models**

rnn\_score = rnn\_model.evaluate(x\_test\_rnn, y\_test, verbose=0)

cnn\_score = cnn\_model.evaluate(x\_test\_cnn, y\_test, verbose=0)

print(f"RNN Accuracy: {rnn\_score[1] \* 100:.2f}%")

print(f"CNN Accuracy: {cnn\_score[1] \* 100:.2f}%")

**# Plot training loss and accuracy**

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(rnn\_history.history['accuracy'], label='RNN Accuracy')

plt.plot(cnn\_history.history['accuracy'], label='CNN Accuracy')

plt.legend()

plt.title('Training Accuracy')

plt.subplot(1, 2, 2)

plt.plot(rnn\_history.history['loss'], label='RNN Loss')

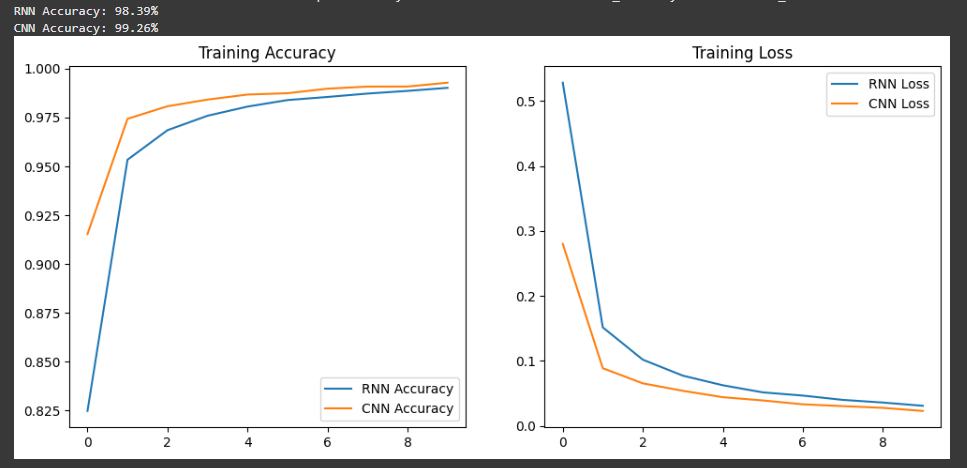
plt.plot(cnn\_history.history['loss'], label='CNN Loss')

plt.legend()

plt.title('Training Loss')

plt.show()

**Output :**

****

**Practical 8**

**Write a program to develop Autoencoders using MNIST Handwritten Digits**

**Code :**

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Flatten, Reshape

from tensorflow.keras.datasets import mnist

**# Load and preprocess the MNIST dataset**

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

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = np.reshape(x\_train, (len(x\_train), 28, 28, 1))

x\_test = np.reshape(x\_test, (len(x\_test), 28, 28, 1))

**# Define the Autoencoder architecture**

input\_img = Input(shape=(28, 28, 1))

**# Encoder**

x = Flatten()(input\_img)

x = Dense(128, activation='relu')(x)

x = Dense(64, activation='relu')(x)

encoded = Dense(32, activation='relu')(x)

**# Decoder**

x = Dense(64, activation='relu')(encoded)

x = Dense(128, activation='relu')(x)

x = Dense(28 \* 28, activation='sigmoid')(x)

decoded = Reshape((28, 28, 1))(x)

**# Define the Autoencoder model**

autoencoder = Model(input\_img, decoded)

**# Compile the model**

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

**# Train the Autoencoder**

autoencoder.fit(

    x\_train, x\_train,

    epochs=50,

    batch\_size=256,

    shuffle=True,

    validation\_data=(x\_test, x\_test))

**# Encode and decode some images**

encoded\_imgs = autoencoder.predict(x\_test)

**# Display original and reconstructed images**

n = 10

plt.figure(figsize=(20, 4))

for i in range(n):

**# Original**

    ax = plt.subplot(2, n, i + 1)

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

    plt.axis('off')

**# Reconstructed**

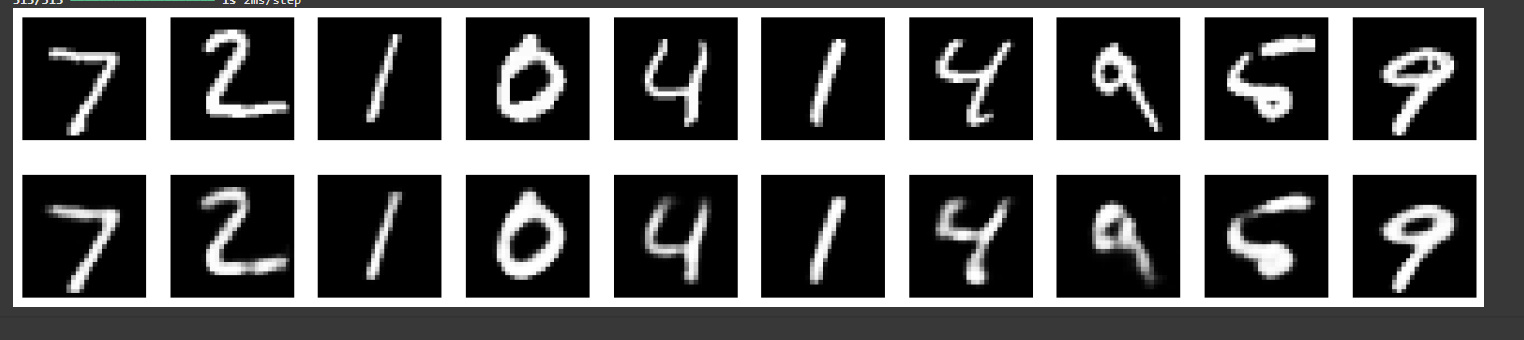
    ax = plt.subplot(2, n, i + 1 + n)

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

    plt.axis('off')

plt.show()

**Output :**

****

**Practical 9**

**Demonstrate recurrent neural network that learns to perform sequence analysis for stock price.(google stock price)**

**Code :**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from sklearn.preprocessing import MinMaxScaler

**# Load Google stock price data**

import kagglehub

from kagglehub import KaggleDatasetAdapter

**# Set the path to the file you'd like to load**

file\_path = "Google\_Stock\_Price\_Train.csv"

**# Load the latest version**

df = kagglehub.load\_dataset(

  KaggleDatasetAdapter.PANDAS,

  "vaibhavsxn/google-stock-prices-training-and-test-data",

  file\_path,

  # Provide any additional arguments like

  # sql\_query or pandas\_kwargs. See the

  # documenation for more information:

  # https://github.com/Kaggle/kagglehub/blob/main/README.md#kaggledatasetadapterpandas)

print("First 5 records:", df.head())

**# Extract the 'Close' prices and preprocess**

stock\_prices = df['Close'].str.replace(',','',regex=True).astype(float).values

stock\_prices = stock\_prices.reshape(-1, 1)

scaler = MinMaxScaler(feature\_range=(0, 1))

scaled\_prices = scaler.fit\_transform(stock\_prices)

**# Prepare data for RNN**

sequence\_length = 60

X, y = [], []

for i in range(len(scaled\_prices) - sequence\_length):

    X.append(scaled\_prices[i:i+sequence\_length])

    y.append(scaled\_prices[i+sequence\_length])

X, y = np.array(X), np.array(y)

**# Reshape X for LSTM input**

X = np.reshape(X, (X.shape[0], X.shape[1], 1))

**# Build RNN model**

model = Sequential([

    LSTM(units=50, return\_sequences=True, input\_shape=(X.shape[1], 1)),

    Dropout(0.2),

    LSTM(units=50, return\_sequences=False),

    Dropout(0.2),

    Dense(units=1)])

**# Compile and train the model**

model.compile(optimizer='adam', loss='mean\_squared\_error')

model.fit(X, y, epochs=50, batch\_size=32)

**# Make predictions**

predicted\_prices = model.predict(X)

predicted\_prices = scaler.inverse\_transform(predicted\_prices)

**# Plot actual vs predicted stock prices**

plt.plot(stock\_prices[sequence\_length:], color='blue', label='Actual Google Stock Price')

plt.plot(predicted\_prices, color='red', label='Predicted Google Stock Price')

plt.title('Google Stock Price Prediction using RNN')

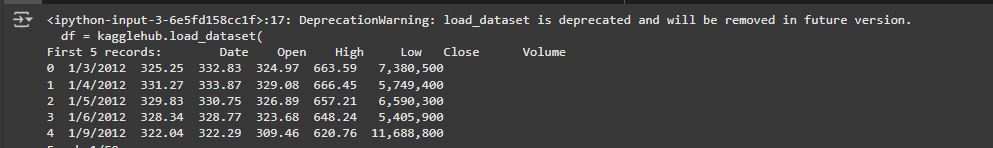
plt.xlabel('Time')

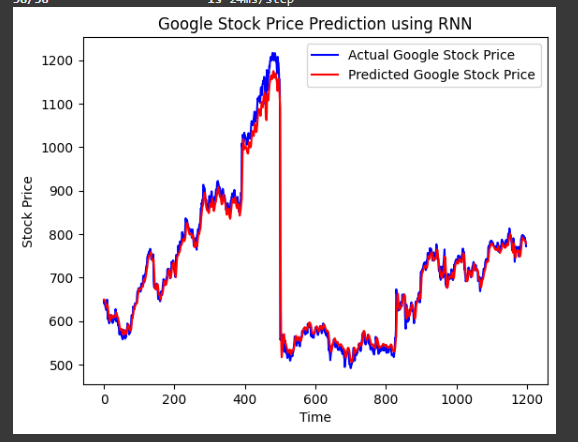
plt.ylabel('Stock Price')

plt.legend()

plt.show()

**Output :**

****

****

**Practical 10**

**Applying Generative Adversarial Networks for image generation and unsupervised tasks.**

**Code :**

import tensorflow as tf

from tensorflow.keras.layers import Dense, Reshape, Flatten, LeakyReLU, Dropout, BatchNormalization, Conv2DTranspose, Conv2D

from tensorflow.keras.models import Sequential

import numpy as np

import matplotlib.pyplot as plt

**# Define the Generator**

def build\_generator():

    model = Sequential()

    model.add(Dense(7 \* 7 \* 256, input\_dim=100))

    model.add(LeakyReLU(0.2))

    model.add(Reshape((7, 7, 256)))

    model.add(Conv2DTranspose(128, kernel\_size=4, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(0.2))

    model.add(Conv2DTranspose(64, kernel\_size=4, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(0.2))

    model.add(Conv2D(1, kernel\_size=7, padding='same', activation='tanh'))

    return model

**# Define the Discriminator**

def build\_discriminator():

    model = Sequential()

    model.add(Conv2D(64, kernel\_size=3, strides=2, padding='same', input\_shape=(28, 28, 1)))

    model.add(LeakyReLU(0.2))

    model.add(Dropout(0.3))

    model.add(Conv2D(128, kernel\_size=3, strides=2, padding='same'))

    model.add(LeakyReLU(0.2))

    model.add(Dropout(0.3))

    model.add(Flatten())

    model.add(Dense(1, activation='sigmoid'))

    return model

**# Compile the GAN**

def build\_gan(generator, discriminator):

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

    discriminator.trainable = False

    model = Sequential([generator, discriminator])

    model.compile(loss='binary\_crossentropy', optimizer='adam')

    return model

**# Load dataset (MNIST for simplicity)**

(X\_train, \_), (\_, \_) = tf.keras.datasets.mnist.load\_data()

X\_train = X\_train / 127.5 - 1.0

X\_train = np.expand\_dims(X\_train, axis=-1)

**# Training loop**

def train\_gan(gan, generator, discriminator, epochs=100, batch\_size=64, sample\_interval=1000):

    valid = np.ones((batch\_size, 1))

    fake = np.zeros((batch\_size, 1))

    for epoch in range(epochs):

        idx = np.random.randint(0, X\_train.shape[0], batch\_size)

        real\_imgs = X\_train[idx]

        noise = np.random.normal(0, 1, (batch\_size, 100))

        fake\_imgs = generator.predict(noise)

        d\_loss\_real = discriminator.train\_on\_batch(real\_imgs, valid)

        d\_loss\_fake = discriminator.train\_on\_batch(fake\_imgs, fake)

        d\_loss = 0.5 \* np.add(d\_loss\_real, d\_loss\_fake)

        noise = np.random.normal(0, 1, (batch\_size, 100))

        g\_loss = gan.train\_on\_batch(noise, valid)

        if epoch % sample\_interval == 0:

            print(f"Epoch {epoch}, D Loss: {d\_loss[0]}, G Loss: {g\_loss}")

            sample\_images(generator)

**# Generate and display images**

def sample\_images(generator, image\_grid\_rows=4, image\_grid\_columns=4):

    noise = np.random.normal(0, 1, (image\_grid\_rows \* image\_grid\_columns, 100))

    gen\_imgs = generator.predict(noise)

    gen\_imgs = 0.5 \* gen\_imgs + 0.5

    fig, axs = plt.subplots(image\_grid\_rows, image\_grid\_columns, figsize=(4, 4), sharex=True, sharey=True)

    count = 0

    for i in range(image\_grid\_rows):

        for j in range(image\_grid\_columns):

            axs[i, j].imshow(gen\_imgs[count, :, :, 0], cmap='gray')

            axs[i, j].axis('off')

            count += 1

    plt.show()

**# Build and compile the GAN**

generator = build\_generator()

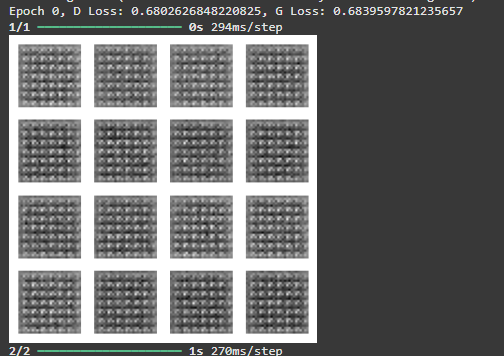
discriminator = build\_discriminator()

gan = build\_gan(generator, discriminator)

**# Train the GAN**

train\_gan(gan, generator, discriminator)

**Output :**

****