|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SR NO** | **PRACTICAL** | **DATE OF PRACTICAL** | **DATE OF SUBMISSION** | **REMARK** |
| **1** | **INTRODUCTION OF TENSOR FLOW** |  |  |  |
| **2** | **LINEAR REGRESSION** |  |  |  |
| **3** | **CONVOLUTIONAL NEURAL NETWORKS** |  |  |  |
| **4** | **WRITE A PROGRAM TO IMPLEMENT DEEP LEARNING TECHNIQUES FOR IMAGE**  **SEGMENTATION** |  |  |  |
| **5** | **WRITE A PROGRAM TO PREDICT A CAPTION FOR A SAMPLE IMAGE USING LSTM** |  |  |  |
| **6** | **APPLYING THE AUTOENCODER ALGORITHM 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 A AUTOENCODER USING MNIST**  **HANDWRITTEN DIGITS** |  |  |  |
| **9** | **DEMONSTARTE RECURRENT NEURAL NETWORK THAT LEARN TO PERFORM SEQUENCE ANALYSIS FOR STOCK** |  |  |  |
| **10** | **APPLYING GENERATIVE ADVERSARIAL NETWORKS FOR IMAGE GENERATION AND UNSURPRISED** |  |  |  |

**PRACTICAL 1**

## INTRODUCTION TO TENSORFLOW

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

## CODE:

import tensorflow as tf

# -------------------- Creating Tensors with Different Shapes -------------------- print("\n--- Creating Tensors with Different Shapes ---")

# 1. Scalar (0-dimensional tensor) scalar\_tensor = tf.constant(10) print("\nScalar Tensor:") print(scalar\_tensor)

print("Shape:", tf.shape(scalar\_tensor))

print("Number of Dimensions (Rank):", tf.rank(scalar\_tensor))

# 2. Vector (1-dimensional tensor) vector\_tensor = tf.constant([1, 2, 3, 4, 5]) print("\nVector Tensor:") print(vector\_tensor)

print("Shape:", tf.shape(vector\_tensor))

print("Number of Dimensions (Rank):", tf.rank(vector\_tensor))

# 3. Matrix (2-dimensional tensor) matrix\_tensor = tf.constant([[1, 2],

[3, 4],

[5, 6]])

print("\nMatrix Tensor:") print(matrix\_tensor)

print("Shape:", tf.shape(matrix\_tensor))

print("Number of Dimensions (Rank):", tf.rank(matrix\_tensor))

# 4. 3-dimensional tensor

tensor\_3d = tf.constant([[[1, 2], [3, 4]],

[[5, 6], [7, 8]]])

print("\n3-dimensional Tensor:") print(tensor\_3d)

print("Shape:", tf.shape(tensor\_3d))

print("Number of Dimensions (Rank):", tf.rank(tensor\_3d))

# 5. Higher-dimensional tensor (e.g., 4-dimensional) tensor\_4d = tf.zeros([2, 3, 2, 5])

print("\n4-dimensional Tensor (initialized with zeros):") print(tensor\_4d)

print("Shape:", tf.shape(tensor\_4d))

print("Number of Dimensions (Rank):", tf.rank(tensor\_4d))

# -------------------- Creating Tensors with Different Data Types -------------------- print("\n--- Creating Tensors with Different Data Types ---")

# 1. Integer data type (default is tf.int32) int\_tensor = tf.constant([1, 2, 3], dtype=tf.int32) print("\nInteger Tensor (tf.int32):") print(int\_tensor)

print("Data Type:", int\_tensor.dtype)

int64\_tensor = tf.constant([1, 2, 3], dtype=tf.int64) print("\nInteger Tensor (tf.int64):") print(int64\_tensor)

print("Data Type:", int64\_tensor.dtype)

# 2. Floating-point data type (default is tf.float32) float\_tensor = tf.constant([1.0, 2.0, 3.0], dtype=tf.float32) print("\nFloating-point Tensor (tf.float32):") print(float\_tensor)

print("Data Type:", float\_tensor.dtype)

float64\_tensor = tf.constant([1.0, 2.0, 3.0], dtype=tf.float64) print("\nFloating-point Tensor (tf.float64):") print(float64\_tensor)

print("Data Type:", float64\_tensor.dtype)

# 3. Boolean data type

bool\_tensor = tf.constant([True, False, True], dtype=tf.bool) print("\nBoolean Tensor:")

print(bool\_tensor)

print("Data Type:", bool\_tensor.dtype)

# 4. String data type

string\_tensor = tf.constant(["hello", "world"], dtype=tf.string) print("\nString Tensor:")

print(string\_tensor)

print("Data Type:", string\_tensor.dtype)

# -------------------- Explicitly Specifying Shape and Data Type Together --------------------

print("\n--- Explicitly Specifying Shape and Data Type Together ---")

shaped\_float\_tensor = tf.constant([[1.5, 2.5], [3.5, 4.5]], dtype=tf.float16) print("\nShaped Float Tensor (tf.float16):")

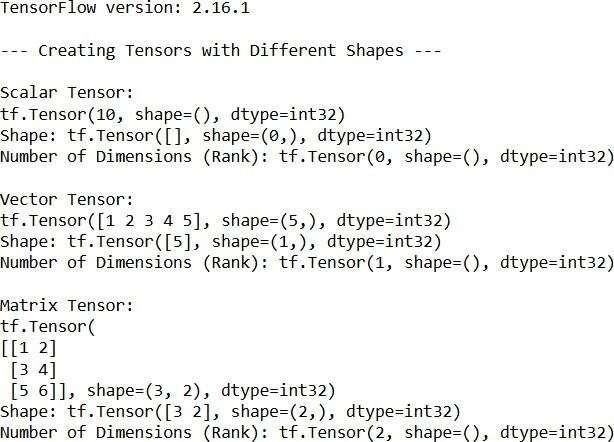
print(shaped\_float\_tensor)

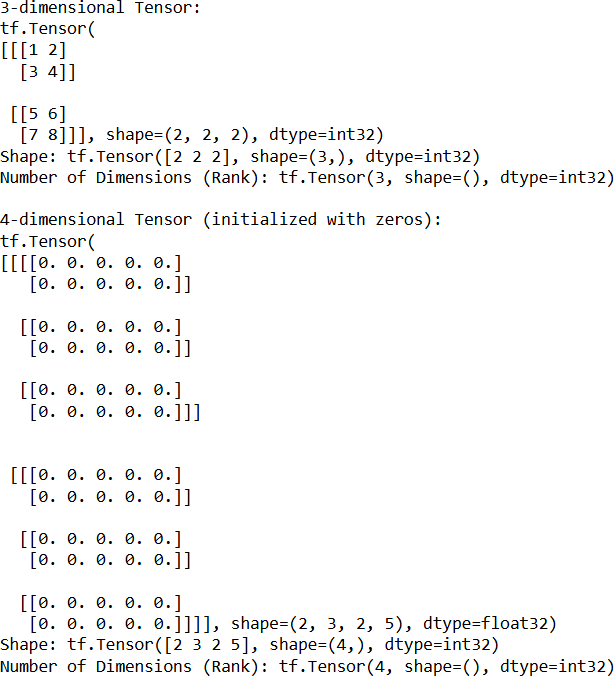
print("Shape:", tf.shape(shaped\_float\_tensor)) print("Data Type:", shaped\_float\_tensor.dtype)

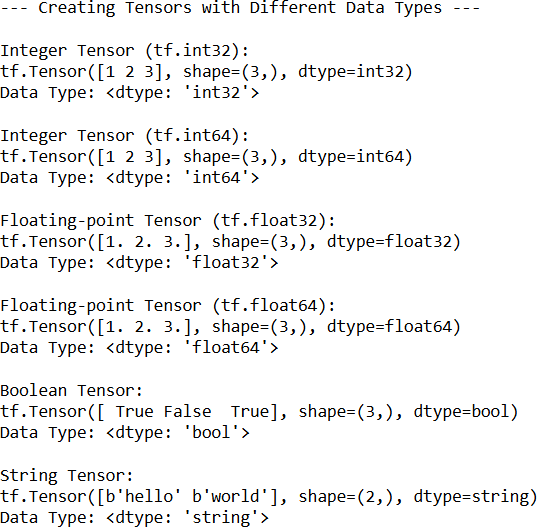
shaped\_int\_tensor = tf.zeros((3, 3), dtype=tf.int32) print("\nShaped Integer Tensor (initialized with zeros, tf.int32):") print(shaped\_int\_tensor)

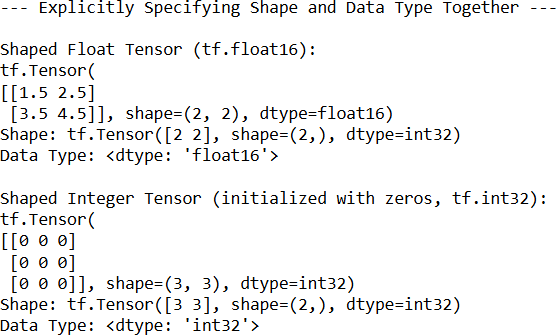
print("Shape:", tf.shape(shaped\_int\_tensor)) print("Data Type:", shaped\_int\_tensor.dtype)

## OUTPUT:



****



****

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

## CODE:

import tensorflow as tf

# Basic Tensor Operations print("--- Basic Tensor Operations ---")

# Create some sample tensors tensor\_a = tf.constant([[1, 2],

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

[7, 8]], dtype=tf.float32) scalar\_c = tf.constant(2.0, dtype=tf.float32)

print("\nTensor A:") print(tensor\_a) print("\nTensor B:") print(tensor\_b) print("\nScalar C:") print(scalar\_c)

# 1. Addition

addition\_result = tf.add(tensor\_a, tensor\_b) print("\nAddition (tensor\_a + tensor\_b):") print(addition\_result)

addition\_scalar = tf.add(tensor\_a, scalar\_c) print("\nAddition with scalar (tensor\_a + scalar\_c):") print(addition\_scalar)

# 2. Subtraction

subtraction\_result = tf.subtract(tensor\_b, tensor\_a) print("\nSubtraction (tensor\_b - tensor\_a):") print(subtraction\_result)

subtraction\_scalar = tf.subtract(tensor\_a, scalar\_c) print("\nSubtraction with scalar (tensor\_a - scalar\_c):") print(subtraction\_scalar)

# 3. Multiplication (Element-wise) multiplication\_result = tf.multiply(tensor\_a, tensor\_b)

print("\nElement-wise Multiplication (tensor\_a \* tensor\_b):") print(multiplication\_result)

multiplication\_scalar = tf.multiply(tensor\_a, scalar\_c) print("\nMultiplication with scalar (tensor\_a \* scalar\_c):") print(multiplication\_scalar)

# 4. Division (Element-wise)

division\_result = tf.divide(tensor\_b, tensor\_a) print("\nElement-wise Division (tensor\_b / tensor\_a):") print(division\_result)

division\_scalar = tf.divide(tensor\_a, scalar\_c) print("\nDivision with scalar (tensor\_a / scalar\_c):") print(division\_scalar)

# Important Considerations print("\n--- Important Considerations ---")

# 1. Shape Compatibility (Broadcasting) tensor\_d = tf.constant([10, 20], dtype=tf.float32) print("\nTensor D:")

print(tensor\_d)

# Addition with a vector (broadcasting) addition\_broadcast = tf.add(tensor\_a, tensor\_d)

print("\nAddition with broadcasting (tensor\_a + tensor\_d):") print(addition\_broadcast)

tensor\_e = tf.constant([[2], [3]], dtype=tf.float32) print("\nTensor E:")

print(tensor\_e)

addition\_broadcast\_2 = tf.add(tensor\_a, tensor\_e) print("\nAddition with broadcasting (tensor\_a + tensor\_e):") print(addition\_broadcast\_2)

# Attempting incompatible shapes will result in an error

# tf.add(tensor\_a, tf.constant([1, 2, 3], dtype=tf.float32)) # This will raise an error

# 2. Data Type Compatibility

tensor\_f = tf.constant([[1, 2], [3, 4]], dtype=tf.int32)

# Attempting operations on tensors with incompatible data types will result in an error # tf.add(tensor\_a, tensor\_f) # This will raise a TypeError

# You need to cast the data type to make them compatible tensor\_f\_float = tf.cast(tensor\_f, tf.float32) addition\_compatible = tf.add(tensor\_a, tensor\_f\_float) print("\nAddition with compatible data types (after casting):") print(addition\_compatible)

# Other Useful Operations print("\n--- Other Useful Operations ---")

# Matrix Multiplication

matrix\_multiplication = tf.matmul(tensor\_a, tensor\_b)

print("\nMatrix Multiplication (tf.matmul(tensor\_a, tensor\_b)):") print(matrix\_multiplication)

# Transpose

tensor\_at = tf.transpose(tensor\_a)

print("\nTranspose of Tensor A (tf.transpose(tensor\_a)):") print(tensor\_at)

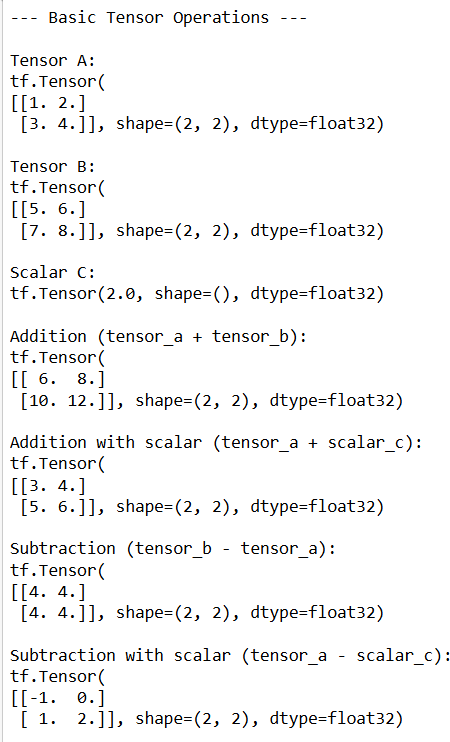
# Element-wise Power power\_result = tf.pow(tensor\_a, 2)

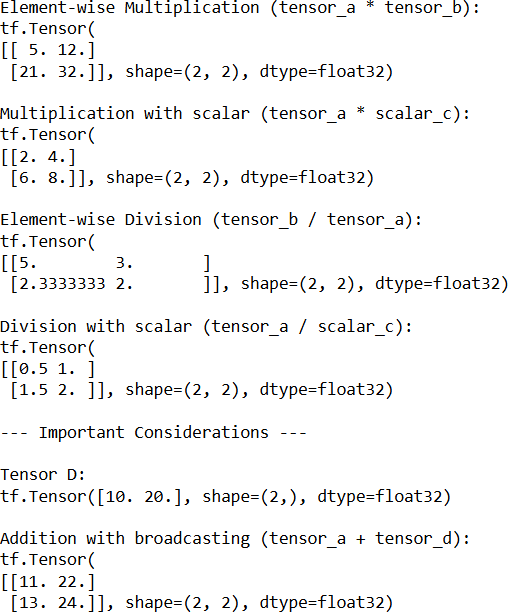
print("\nElement-wise Power (tf.pow(tensor\_a, 2)):") print(power\_result)

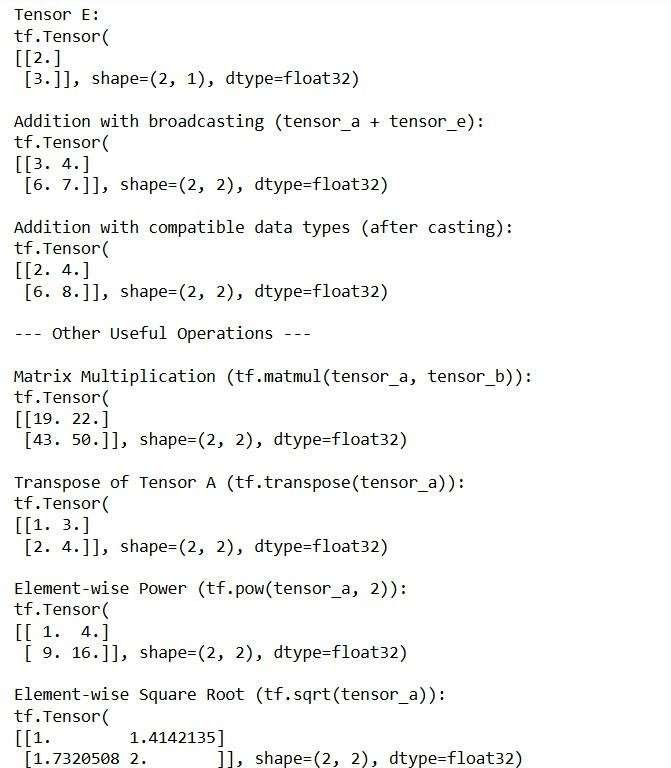
# Square Root (element-wise) sqrt\_result = tf.sqrt(tensor\_a)

print("\nElement-wise Square Root (tf.sqrt(tensor\_a)):") print(sqrt\_result)

## OUTPUT:

****





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

## CODE:

import tensorflow as tf import numpy as np

# 1. Create a sample tensor tensor = tf.constant([

[[1, 2, 3], [4, 5, 6]],

[[7, 8, 9], [10, 11, 12]],

[[13, 14, 15], [16, 17, 18]]

], dtype=tf.int32)

print("Original Tensor (shape: {}):\n{}".format(tensor.shape, tensor.numpy()))

# Reshaping

print("\n--- Reshaping ---")

# Reshape to a 2D tensor

reshaped\_tensor\_2d = tf.reshape(tensor, [2, 9])

print("\nReshaped to 2x9 (shape: {}):\n{}".format(reshaped\_tensor\_2d.shape, reshaped\_tensor\_2d.numpy()))

# Reshape to a 1D tensor

reshaped\_tensor\_1d = tf.reshape(tensor, [-1]) # -1 infers the size

print("\nReshaped to 1D (shape: {}):\n{}".format(reshaped\_tensor\_1d.shape, reshaped\_tensor\_1d.numpy()))

# Reshape to a different 3D shape reshaped\_tensor\_3d = tf.reshape(tensor, [1, 3, 6])

print("\nReshaped to 1x3x6 (shape: {}):\n{}".format(reshaped\_tensor\_3d.shape, reshaped\_tensor\_3d.numpy()))

# Slicing

print("\n--- Slicing ---")

# Basic slicing (similar to Python lists/NumPy arrays) slice\_row\_0 = tensor[0, :, :] # First "outer" dimension

print("\nSlice: First 'outer' dimension (shape: {}):\n{}".format(slice\_row\_0.shape, slice\_row\_0.numpy()))

slice\_col\_1 = tensor[:, :, 1] # All "outer", all "middle", second element of "inner"

print("\nSlice: Second element of the inner dimension (shape:

{}):\n{}".format(slice\_col\_1.shape, slice\_col\_1.numpy()))

slice\_specific = tensor[1, 0, 2] # Element at index [1, 0, 2]

print("\nSlice: Element at [1, 0, 2]: {}".format(slice\_specific.numpy()))

# Using start:stop:step

slice\_range = tensor[0:2, 0:1, :] # First two "outer", first "middle", all "inner"

print("\nSlice with range (shape: {}):\n{}".format(slice\_range.shape, slice\_range.numpy()))

slice\_step = tensor[:, :, ::2] # All "outer", all "middle", every other element of "inner" print("\nSlice with step (shape: {}):\n{}".format(slice\_step.shape, slice\_step.numpy()))

# Indexing

print("\n--- Indexing ---")

# Accessing a single element element = tensor[2, 1, 0]

print("\nElement at [2, 1, 0]: {}".format(element.numpy()))

# Using integer arrays for indexing (gather\_nd) indices = [[0, 0, 0], [1, 1, 2], [2, 0, 1]]

indexed\_elements = tf.gather\_nd(tensor, indices)

print("\nIndexed elements using gather\_nd (indices {}):\n{}".format(indices, indexed\_elements.numpy()))

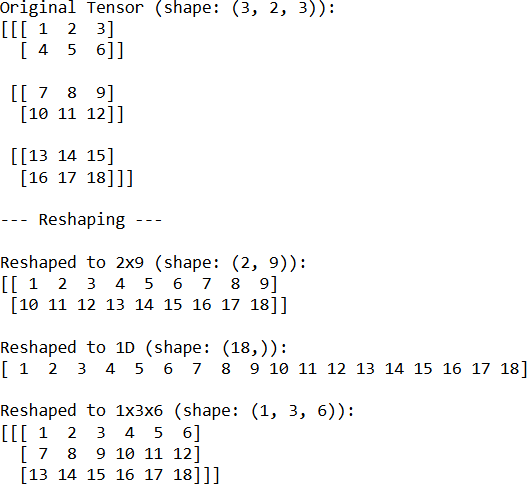
# Using boolean masks for indexing (boolean indexing) mask = tensor > 10

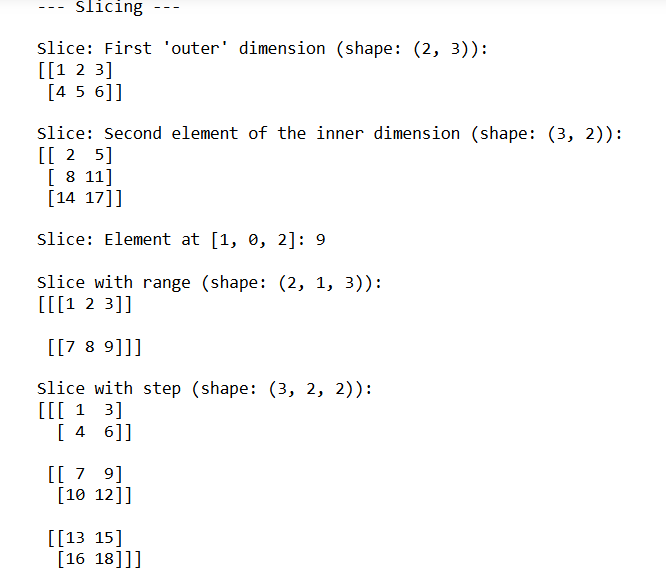
print("\nBoolean mask (tensor > 10):\n{}".format(mask.numpy()))

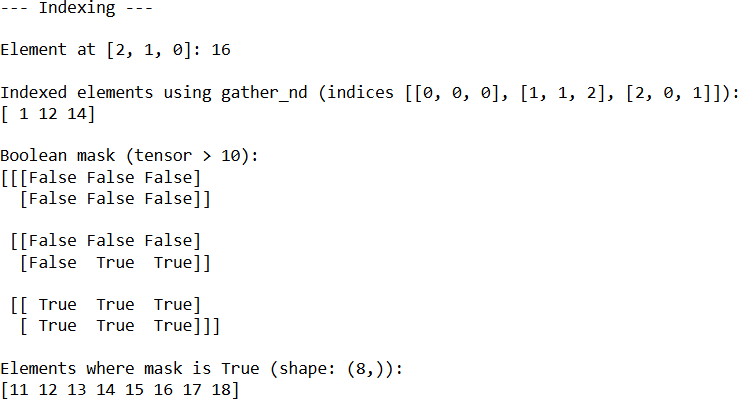
masked\_elements = tf.boolean\_mask(tensor, mask)

print("\nElements where mask is True (shape: {}):\n{}".format(masked\_elements.shape, masked\_elements.numpy()))

## OUTPUT:

****



****

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

## CODE:

import tensorflow as tf

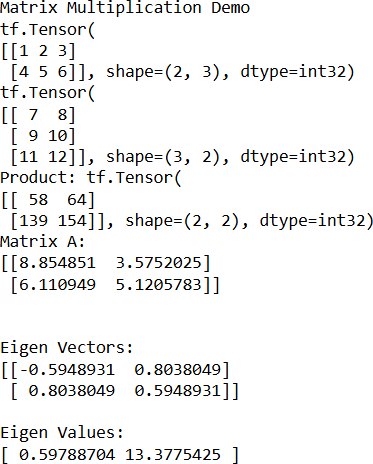
print("Matrix Multiplication Demo") x=tf.constant([1,2,3,4,5,6],shape=[2,3]) print(x) y=tf.constant([7,8,9,10,11,12],shape=[3,2]) print(y)

z=tf.matmul(x,y) print("Product:",z)

e\_matrix\_A=tf.random.uniform([2,2],minval=3,maxval=10,dtype=tf.float32,name="matrixA") print("Matrix A:\n{}\n\n".format(e\_matrix\_A)) eigen\_values\_A,eigen\_vectors\_A=tf.linalg.eigh(e\_matrix\_A)

print("Eigen Vectors:\n{}\n\nEigen Values:\n{}\n".format(eigen\_vectors\_A,eigen\_values\_A))

## OUTPUT:

****

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

## CODE:

import numpy as np

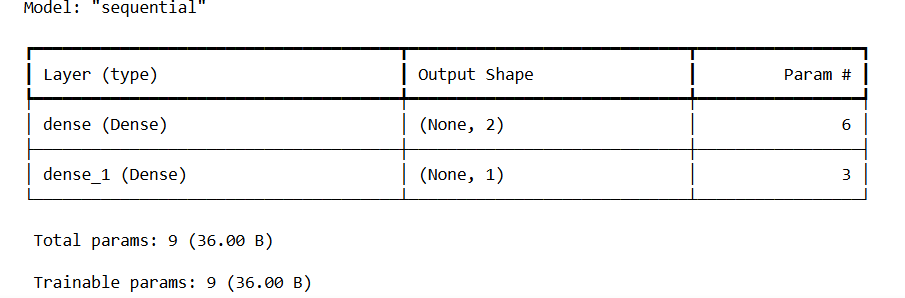
from keras.layers import Dense from keras.models import Sequential model=Sequential()

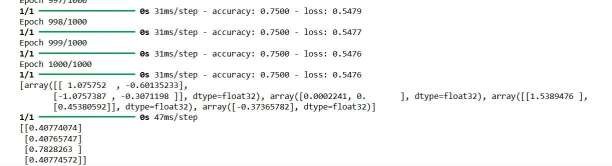
model.add(Dense(units=2,activation='relu',input\_dim=2)) model.add(Dense(units=1,activation='sigmoid')) model.compile(loss='binary\_crossentropy',optimizer='adam',metrics=['accuracy']) print(model.summary())

print(model.get\_weights()) X=np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]]) Y=np.array([0.,1.,1.,0.])

model.fit(X,Y,epochs=1000,batch\_size=4) print(model.get\_weights()) print(model.predict(X,batch\_size=4))

## OUTPUT:



****

**Linear Regression**

# PRACTICAL 2

1. **1. Implement a simple linear regression model using TensorFlow's low level API (or tf. keras).**
2. **Train the model on a toy dataset (e.g., housing prices vs. square footage).**
3. **Visualize the loss function and the learned linear relationship.**
4. **Make predictions on new data points.**

## CODE:

import tensorflow as tf import numpy as np

import matplotlib.pyplot as plt

# Generate a toy dataset np.random.seed(42) num\_samples = 100

square\_footage = np.random.uniform(500, 2000, num\_samples).astype(np.float32)

true\_price = 50 + 150 \* square\_footage + np.random.normal(0, 5000, num\_samples).astype(np.float32)

# Normalize the features (important for training stability) mean\_sqft = np.mean(square\_footage)

std\_sqft = np.std(square\_footage)

normalized\_sqft = (square\_footage - mean\_sqft) / std\_sqft

# Define the model parameters (TensorFlow Variables)

W = tf.Variable(np.random.randn(), name="weight", dtype=tf.float32) b = tf.Variable(0.0, name="bias", dtype=tf.float32)

# Define the linear regression model def linear\_regression(x):

return W \* x + b

# Define the loss function (Mean Squared Error) def mean\_squared\_error(y\_pred, y\_true):

return tf.reduce\_mean(tf.square(y\_pred - y\_true))

# Define the optimizer learning\_rate = 0.01

optimizer = tf.optimizers.SGD(learning\_rate)

# Training loop epochs = 100 loss\_history = []

for epoch in range(epochs):

with tf.GradientTape() as tape:

predictions = linear\_regression(normalized\_sqft) loss = mean\_squared\_error(predictions, true\_price)

# Calculate gradients

gradients = tape.gradient(loss, [W, b])

# Update model parameters optimizer.apply\_gradients(zip(gradients, [W, b]))

loss\_history.append(loss.numpy()) if (epoch + 1) % 10 == 0:

print(f"Epoch {epoch + 1}, Loss: {loss.numpy():.4f}, Weight: {W.numpy():.2f}, Bias:

{b.numpy():.2f}")

# Visualize the loss function plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)

plt.plot(range(1, epochs + 1), loss\_history) plt.title('Loss Function') plt.xlabel('Epoch')

plt.ylabel('Mean Squared Error')

# Visualize the learned linear relationship plt.subplot(1, 2, 2)

plt.scatter(normalized\_sqft, true\_price, label='Training Data') predicted\_prices = linear\_regression(normalized\_sqft)

plt.plot(normalized\_sqft, predicted\_prices, 'r-', label='Learned Regression Line') plt.title('Learned Linear Relationship')

plt.xlabel('Normalized Square Footage') plt.ylabel('Price')

plt.legend() plt.tight\_layout() plt.show()

# Make predictions on new data points

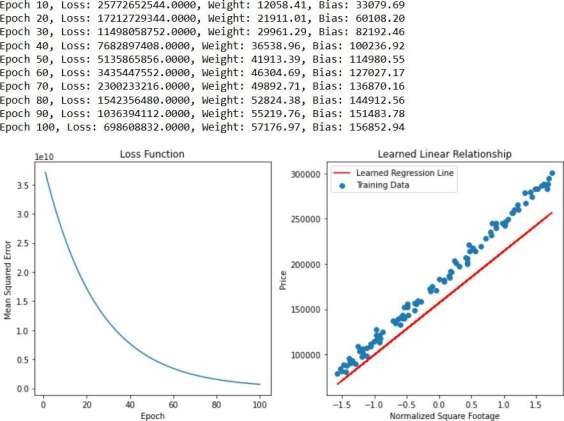
new\_square\_footage = np.array([750, 1500, 2200]).astype(np.float32) normalized\_new\_sqft = (new\_square\_footage - mean\_sqft) / std\_sqft predicted\_prices\_new = linear\_regression(normalized\_new\_sqft).numpy()

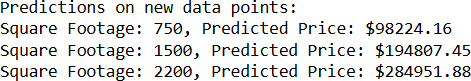
print("\nPredictions on new data points:") for i in range(len(new\_square\_footage)):

print(f"Square Footage: {new\_square\_footage[i]:.0f}, Predicted Price:

${predicted\_prices\_new[i]:.2f}")

**OUTPUT:**

****

****

# PRACTICAL 3

**Convolutional Neural Networks (Classification)**

1. **Implementing deep neural network for performing binary classification task CODE:**

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split from sklearn.preprocessing import StandardScaler from sklearn.datasets import make\_classification import numpy as np

import matplotlib.pyplot as plt

# 1. Generate a synthetic binary classification dataset

X, y = make\_classification(n\_samples=1000, n\_features=20, n\_informative=15, n\_redundant=5, random\_state=42)

# 2. Split the dataset into training and testing sets

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

# 3. Feature Scaling (important for neural networks) scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

# 4. Define the Deep Neural Network model using TensorFlow Keras model = tf.keras.Sequential([

tf.keras.layers.Dense(units=64, activation='relu', input\_shape=(X\_train\_scaled.shape[1],)), tf.keras.layers.Dense(units=32, activation='relu'),

tf.keras.layers.Dense(units=1, activation='sigmoid') # Output layer for binary classification

])

# 5. Compile the model model.compile(optimizer='adam',

loss='binary\_crossentropy', # Suitable loss for binary classification metrics=['accuracy'])

# 6. Train the model

history = model.fit(X\_train\_scaled, y\_train, epochs=50, batch\_size=32, validation\_split=0.1, verbose=0)

# 7. Evaluate the model on the test set

loss, accuracy = model.evaluate(X\_test\_scaled, y\_test, verbose=0) print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")

# 8. Make predictions on new data

# Let's take the first 5 samples from the test set as new data new\_data = X\_test\_scaled[:5]

predictions = model.predict(new\_data)

predicted\_classes = (predictions > 0.5).astype(int) # Convert probabilities to binary classes

print("\nPredictions on new data:") for i in range(len(new\_data)):

print(f"Sample: {i+1}, Predicted Probability: {predictions[i][0]:.4f}, Predicted Class:

{predicted\_classes[i][0]}, True Class: {y\_test[i]}")

# 9. Visualize training history (optional) plt.figure(figsize=(12, 4))

# Plot training & validation accuracy values plt.subplot(1, 2, 1) plt.plot(history.history['accuracy']) plt.plot(history.history['val\_accuracy']) plt.title('Model accuracy') plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper left')

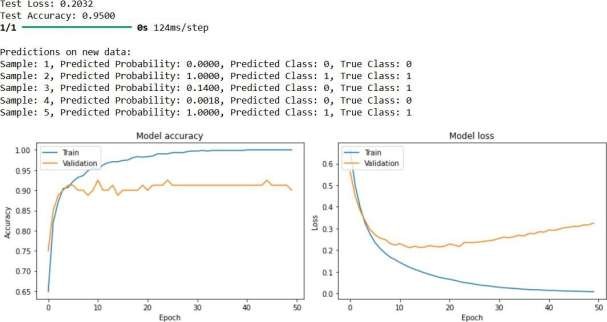
# Plot training & validation loss values plt.subplot(1, 2, 2) plt.plot(history.history['loss']) plt.plot(history.history['val\_loss']) plt.title('Model loss')

plt.ylabel('Loss') plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout() plt.show()

## OUTPUT:

****

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

## CODE:

#pract3 b

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split from sklearn.preprocessing import StandardScaler from sklearn.datasets import load\_iris

from sklearn.metrics import classification\_report, confusion\_matrix import numpy as np

import matplotlib.pyplot as plt import seaborn as sns

# 1. Load the Iris dataset (a classic multiclass classification dataset) iris = load\_iris()

X = iris.data y = iris.target

class\_names = iris.target\_names

# 2. Split the dataset into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# 3. Feature Scaling scaler = StandardScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

# 4. Define the Deep Feed-Forward Neural Network model model = tf.keras.Sequential([

tf.keras.layers.Dense(units=64, activation='relu', input\_shape=(X\_train\_scaled.shape[1],)), #

First hidden layer

tf.keras.layers.Dense(units=32, activation='relu'), # Second hidden layer

tf.keras.layers.Dense(units=len(class\_names), activation='softmax') # Output layer

])

# 5. Compile the model model.compile(optimizer='adam',

loss='sparse\_categorical\_crossentropy', # Suitable loss for integer-encoded multiclass metrics=['accuracy'])

# 6. Train the model

history = model.fit(X\_train\_scaled, y\_train, epochs=100, batch\_size=32, validation\_split=0.1, verbose=0)

# 7. Evaluate the model on the test set

loss, accuracy = model.evaluate(X\_test\_scaled, y\_test, verbose=0) print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")

# 8. Make predictions on the test set predictions\_probabilities = model.predict(X\_test\_scaled)

predicted\_classes = np.argmax(predictions\_probabilities, axis=1)

# 9. Print classification report and confusion matrix print("\nClassification Report:")

print(classification\_report(y\_test, predicted\_classes, target\_names=class\_names))

cm = confusion\_matrix(y\_test, predicted\_classes) plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

xticklabels=class\_names, yticklabels=class\_names) plt.xlabel('Predicted Class')

plt.ylabel('True Class') plt.title('Confusion Matrix') plt.show()

# 10. Visualize training history (optional) plt.figure(figsize=(12, 4))

# Plot training & validation accuracy values plt.subplot(1, 2, 1) plt.plot(history.history['accuracy']) plt.plot(history.history['val\_accuracy']) plt.title('Model accuracy') plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper left')

# Plot training & validation loss values plt.subplot(1, 2, 2) plt.plot(history.history['loss']) plt.plot(history.history['val\_loss']) plt.title('Model loss')

plt.ylabel('Loss') plt.xlabel('Epoch')

plt.legend(['Train', 'Validation'], loc='upper left')

plt.tight\_layout() plt.show()

# 11. Predict the class for a new, unseen data point

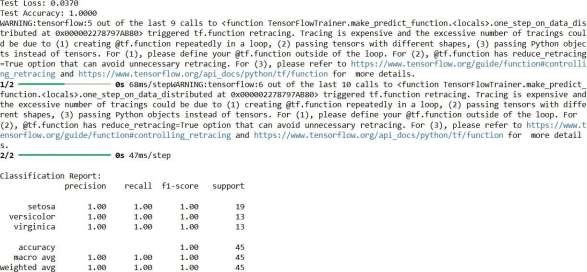
new\_data\_point = np.array([[5.1, 3.5, 1.4, 0.2]]) # Example feature values new\_data\_scaled = scaler.transform(new\_data\_point) # Remember to scale! prediction\_probabilities\_new = model.predict(new\_data\_scaled) predicted\_class\_index\_new = np.argmax(prediction\_probabilities\_new, axis=1)[0] predicted\_class\_name\_new = class\_names[predicted\_class\_index\_new]

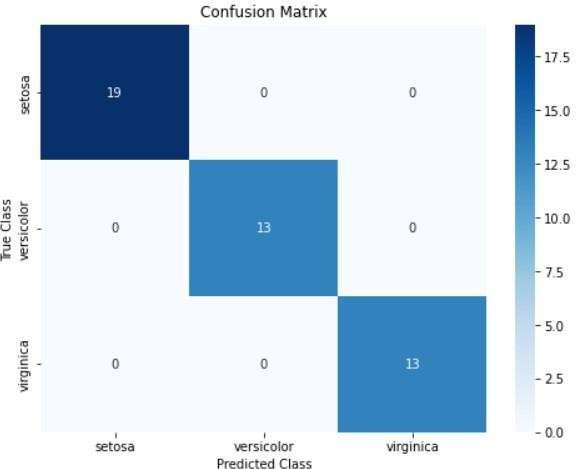
print(f"\nPrediction for new data point [{new\_data\_point[0][0]:.1f}, {new\_data\_point[0][1]:.1f},

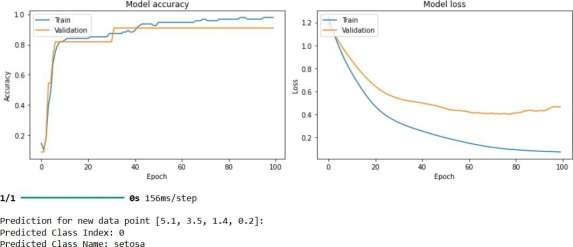
{new\_data\_point[0][2]:.1f}, {new\_data\_point[0][3]:.1f}]:") print(f"Predicted Class Index: {predicted\_class\_index\_new}")

print(f"Predicted Class Name: {predicted\_class\_name\_new}")

**OUTPUT:**

****

****



# PRACTICAL 4

**Write a program to implement deep learning techniques for image segmentation CODE:**

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers import numpy as np

import matplotlib.pyplot as plt import os

# 1. Load and Preprocess the Dataset

def generate\_dummy\_data(num\_samples, image\_size): """

Generates dummy image and mask data for demonstration purposes.

Args:

num\_samples: The number of samples to generate. image\_size: The size of the images (e.g., (128, 128)).

Returns:

A tuple containing:

* images: A NumPy array of shape (num\_samples, image\_size[0], image\_size[1], 3) representing the dummy images.
* masks: A NumPy array of shape (num\_samples, image\_size[0], image\_size[1], 1) representing the dummy segmentation masks (0 or 1 values).

"""

images = np.random.randint(0, 256, size=(num\_samples, image\_size[0], image\_size[1], 3), dtype=np.uint8)

masks = np.random.randint(0, 2, size=(num\_samples, image\_size[0], image\_size[1], 1), dtype=np.uint8)

return images, masks

# Example usage:

image\_size = (128, 128) # Define the image size num\_samples = 1000

images, masks = generate\_dummy\_data(num\_samples, image\_size)

# Split the data into training and testing sets train\_ratio = 0.8

train\_size = int(train\_ratio \* num\_samples)

train\_images, train\_masks = images[:train\_size], masks[:train\_size] test\_images, test\_masks = images[train\_size:], masks[train\_size:]

# Normalize the images (important for neural networks) train\_images = train\_images / 255.0

test\_images = test\_images / 255.0

# 2. Define the Model (a simple U-Net)

# U-Net is a popular architecture for image segmentation. This is a simplified version. def create\_unet\_model(image\_size):

"""

Creates a simplified U-Net model for image segmentation.

Args:

image\_size: The size of the input images (height, width). Assumes square images.

Returns:

A Keras model. """

inputs = keras.Input(shape=(image\_size[0], image\_size[1], 3))

# Encoder

conv1 = layers.Conv2D(32, 3, activation='relu', padding='same')(inputs) conv1 = layers.Conv2D(32, 3, activation='relu', padding='same')(conv1) pool1 = layers.MaxPooling2D(pool\_size=(2, 2))(conv1)

conv2 = layers.Conv2D(64, 3, activation='relu', padding='same')(pool1) conv2 = layers.Conv2D(64, 3, activation='relu', padding='same')(conv2) pool2 = layers.MaxPooling2D(pool\_size=(2, 2))(conv2)

# Bottleneck

conv3 = layers.Conv2D(128, 3, activation='relu', padding='same')(pool2) conv3 = layers.Conv2D(128, 3, activation='relu', padding='same')(conv3)

# Decoder

up4 = layers.UpSampling2D(size=(2, 2))(conv3) concat4 = layers.Concatenate()([up4, conv2])

conv4 = layers.Conv2D(64, 3, activation='relu', padding='same')(concat4) conv4 = layers.Conv2D(64, 3, activation='relu', padding='same')(conv4)

up5 = layers.UpSampling2D(size=(2, 2))(conv4) concat5 = layers.Concatenate()([up5, conv1])

conv5 = layers.Conv2D(32, 3, activation='relu', padding='same')(concat5) conv5 = layers.Conv2D(32, 3, activation='relu', padding='same')(conv5)

# Output layer

outputs = layers.Conv2D(1, 1, activation='sigmoid')(conv5) # Use sigmoid for binary segmentation

return keras.Model(inputs=inputs, outputs=outputs)

model = create\_unet\_model(image\_size) model.summary() # Print the model architecture

# 3. Compile the Model

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

# 4. Train the Model

epochs = 10 # Adjust as needed batch\_size = 32

history = model.fit(train\_images, train\_masks, epochs=epochs, batch\_size=batch\_size,

validation\_data=(test\_images, test\_masks))

# 5. Evaluate the Model

loss, accuracy = model.evaluate(test\_images, test\_masks, verbose=0) print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")

# 6. Make Predictions and Visualize Results

def display\_predictions(display\_list, model\_name=""): """Displays a list of images, masks and predicted masks.""" plt.figure(figsize=(15, 15))

title = ['Input Image', 'True Mask', 'Predicted Mask']

for i in range(len(display\_list)): plt.subplot(1, len(display\_list), i+1) plt.title(title[i])

plt.imshow(tf.keras.utils.array\_to\_img(display\_list[i])) plt.axis('off')

plt.tight\_layout() plt.show()

def create\_mask(pred\_mask):

""" Returns a mask with shape [image\_size, 1] Args:

pred\_mask: a tensor of shape [image\_size, num\_classes] with the mask """

pred\_mask = tf.argmax(pred\_mask, axis=-1) pred\_mask = pred\_mask[..., tf.newaxis] return pred\_mask

def show\_predictions(dataset=None, num=1):

"""

Displays the first num images of the dataset and their predicted masks.

Args:

dataset (tf.data.Dataset): The dataset to display predictions from.

If None, uses the test dataset.

num (int): The number of predictions to display.

"""

if dataset:

for image, mask in dataset.take(num):

pred\_mask = model.predict(image)

display\_predictions([image[0], mask[0], create\_mask(pred\_mask)[0]]) else:

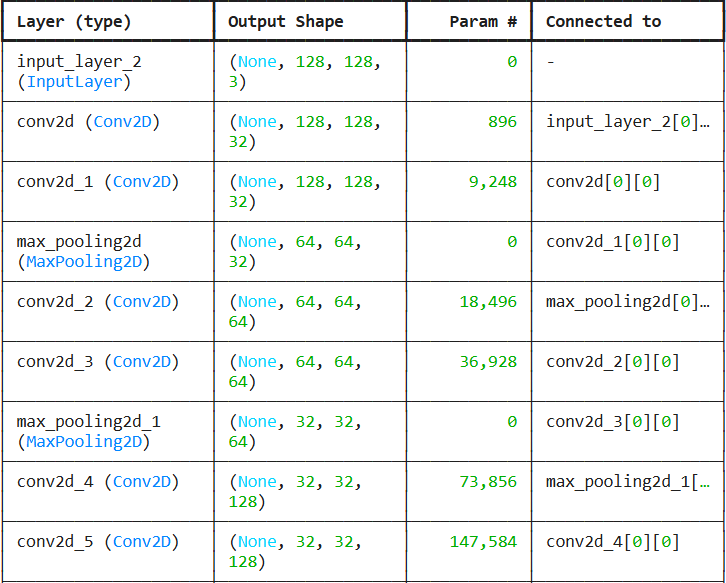
for i in range(num):

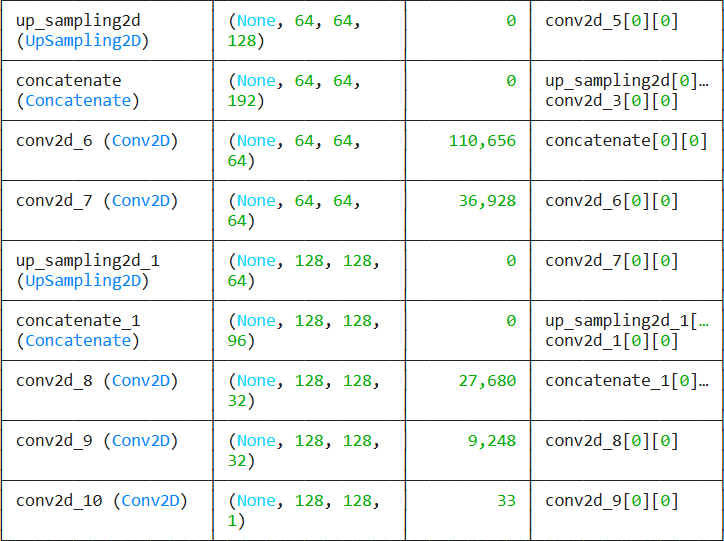
image = test\_images[i] mask = test\_masks[i]

pred\_mask = model.predict(tf.expand\_dims(image, axis=0)) display\_predictions([image, mask, create\_mask(pred\_mask)[0]], model\_name)

show\_predictions(num=5) # Show predictions for the first 5 test images # 7. Optional: Save the Model

# 8. Optional: Load the model.

# loaded\_model = keras.models.load\_model('image\_segmentation\_model.h5') OUTPUT:





# PRACTICAL 5

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

## CODE:

import tensorflow as tf

from tensorflow.keras.applications import VGG16

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

from tensorflow.keras.layers import Input, Dense, LSTM, Embedding, Dropout, Add from tensorflow.keras.preprocessing.sequence import pad\_sequences

import numpy as np

import matplotlib.pyplot as plt

from PIL import Image # Used for image loading and display

# --- 1. Configuration and Dummy Data ---

# Define image input shape for VGG16 IMG\_SHAPE = (224, 224)

word\_to\_idx = {

'<start>': 0, 'a': 1, 'cat': 2, 'dog': 3, 'is': 4, 'running': 5,

'playing': 6, 'in': 7, 'the': 8, 'park': 9, 'house': 10, '<end>': 11

}

idx\_to\_word = {idx: word for word, idx in word\_to\_idx.items()} VOCAB\_SIZE = len(word\_to\_idx)

MAX\_CAPTION\_LENGTH = 15 # Maximum length of a generated caption # --- 2. Image Preprocessing and Feature Extraction (Encoder) ---

def preprocess\_image(image\_path): try:

img = load\_img(image\_path, target\_size=IMG\_SHAPE) img = img\_to\_array(img)

img = np.expand\_dims(img, axis=0) # Add batch dimension

img = tf.keras.applications.vgg16.preprocess\_input(img) # VGG16 specific preprocessing return img

except Exception as e:

print(f"Error loading or preprocessing image: {e}") return None

def extract\_image\_features(image\_array):

vgg\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(IMG\_SHAPE[0], IMG\_SHAPE[1], 3))

# Create a new model that outputs the features

feature\_extractor = Model(inputs=vgg\_model.input, outputs=vgg\_model.layers[-1].output) features = feature\_extractor.predict(image\_array)

# Reshape features to a 2D vector (flattening the spatial dimensions) features = features.reshape(features.shape[0], -1)

return features

# --- 3. Define the LSTM Captioning Model (Decoder) ---

def define\_captioning\_model(vocab\_size, max\_caption\_length, embedding\_dim=256, lstm\_units=512):

image\_features\_input = Input(shape=(25088,)) # Adjust this shape based on actual VGG16 output

# Project image features to a dimension compatible with LSTM image\_features\_dense = Dense(lstm\_units, activation='relu')(image\_features\_input) image\_features\_dropout = Dropout(0.5)(image\_features\_dense)

text\_input = Input(shape=(max\_caption\_length,))

# Word Embedding layer: converts word IDs to dense vectors

text\_embedding = Embedding(vocab\_size, embedding\_dim, mask\_zero=True)(text\_input) text\_dropout = Dropout(0.5)(text\_embedding)

# LSTM layer processes the sequence text\_lstm = LSTM(lstm\_units)(text\_dropout)

decoder\_output = Add()([image\_features\_dropout, text\_lstm]) output = Dense(vocab\_size, activation='softmax')(decoder\_output)

# Create the model

model = Model(inputs=[image\_features\_input, text\_input], outputs=output) model.compile(loss='categorical\_crossentropy', optimizer='adam')

return model

# --- 4. Caption Generation (Inference) ---

def generate\_caption(model, image\_features, word\_to\_idx, idx\_to\_word, max\_caption\_length): # Start the caption with '<start>' token

in\_text = '<start>'

for i in range(max\_caption\_length):

# Convert the current sequence of words to numerical IDs

sequence = [word\_to\_idx[word] for word in in\_text.split() if word in word\_to\_idx] # Pad the sequence to the maximum caption length

sequence = pad\_sequences([sequence], maxlen=max\_caption\_length, padding='post')[0] sequence = np.expand\_dims(sequence, axis=0) # Add batch dimension

# Predict the next word

yhat = model.predict([image\_features, sequence], verbose=0) # Get the index of the word with the highest probability yhat\_idx = np.argmax(yhat)

# Map the index back to a word

word = idx\_to\_word.get(yhat\_idx, None) # Use .get to handle OOV words gracefully

# If word is None (out of vocabulary) or '<end>' token, stop if word is None or word == '<end>':

break

# Append the predicted word to the sequence in\_text += ' ' + word

final\_caption = in\_text.replace('<start>', '').replace('<end>', '').strip() return final\_caption

# --- 5. Main Execution Flow (Simulated in Jupyter) ---

if name == 'main':

print("--- Image Captioning Model Demonstration ---")

print("This is a conceptual example. No actual training is performed.")

dummy\_image\_path = "dummy\_image.jpg" try:

# Create a simple white image for demonstration

dummy\_img = Image.new('RGB', IMG\_SHAPE, color = 'white') dummy\_img.save(dummy\_image\_path)

print(f"Created a dummy image at: {dummy\_image\_path}") except Exception as e:

print(f"Could not create dummy image: {e}. Please ensure you have Pillow installed.") print("You will need to manually provide an image path for the next steps.") dummy\_image\_path = None # Set to None if creation failed

if dummy\_image\_path:

# Step 1: Load and Preprocess a Sample Image

print(f"\n1. Loading and preprocessing image: {dummy\_image\_path}") sample\_image\_array = preprocess\_image(dummy\_image\_path)

if sample\_image\_array is not None:

# Step 2: Extract Features using VGG16

print("2. Extracting image features using VGG16...") sample\_image\_features = extract\_image\_features(sample\_image\_array) print(f" Extracted features shape: {sample\_image\_features.shape}")

# Step 3: Define the Captioning Model

print("\n3. Defining the LSTM captioning model architecture...") captioning\_model = define\_captioning\_model(VOCAB\_SIZE,

MAX\_CAPTION\_LENGTH)

captioning\_model.summary() # Print model summary

# Step 4: Generate a Caption for the Sample Image

print("\n4. Generating a caption for the sample image (using dummy model output)...")

predicted\_caption = generate\_caption( captioning\_model, sample\_image\_features, word\_to\_idx,

idx\_to\_word, MAX\_CAPTION\_LENGTH

)

print(f"\nPredicted Caption: \"{predicted\_caption}\"")

# Display the dummy image plt.figure(figsize=(6, 6)) plt.imshow(Image.open(dummy\_image\_path)) plt.title("Sample Image")

plt.axis('off') plt.show()

else:

print("Could not proceed with feature extraction and caption generation due to image loading error.")

else:

print("Skipping image processing and caption generation because dummy image could not be created.")

print("Please manually create a 'dummy\_image.jpg' or provide a valid path to an image.")

print("\n--- End of Demonstration ---")

print("Remember to train a real model with a proper dataset for meaningful captions.")

## OUTPUT:

**--- Image Captioning Model Demonstration ---**

**This is a conceptual example. No actual training is performed. Created a dummy image at: dummy\_image.jpg**

1. **Loading and preprocessing image: dummy\_image.jpg**
2. **Extracting image features using VGG16...**

**1/1** ━━━━━━━━━━━━━━━━━━━━ **1s 809ms/step Extracted features shape: (1, 25088)**

1. **Defining the LSTM captioning model architecture...**

**Model: "functional\_3"**

┏━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━

━━━┳━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃ **Connected to** ┃

┡━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━

━━━╇━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━┩

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **│ input\_layer\_5** | **│ (None, 15)** |  | **│** |  | **0 │ -** |  | **│** |
| **│ (InputLayer)** | **│** | **│** |  | **│** |  | **│** |  |

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **│ input\_layer\_4** | **│ (None, 25088)** | **│** |  | **0 │ -** |  | **│** |
| **│ (InputLayer)** | **│ │** |  | **│** |  | **│** |  |

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ embedding\_1 │ (None, 15, 256) │ 3,072 │ input\_layer\_5[0]… │**

**│ (Embedding) │ │ │ │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ dense\_2 (Dense) │ (None, 512) │ 12,845,568 │ input\_layer\_4[0]… │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ dropout\_3 (Dropout) │ (None, 15, 256) │ 0 │ embedding\_1[0][0] │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **│ not\_equal\_1** | **│ (None, 15)** |  | **│** |  | **0 │ input\_layer\_5[0]… │** |
| **│ (NotEqual)** | **│** | **│** |  | **│** | **│** |

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ dropout\_2 (Dropout) │ (None, 512) │ 0 │ dense\_2[0][0] │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ lstm\_1 (LSTM) │ (None, 512) │ 1,574,912 │ dropout\_3[0][0], │**

**│ │ │ │ not\_equal\_1[0][0] │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ add\_1 (Add) │ (None, 512) │ 0 │ dropout\_2[0][0], │**

**│ │ │ │ lstm\_1[0][0] │**

**├─────────────────────┼───────────────────┼────────────**

**┼───────────────────┤**

**│ dense\_3 (Dense) │ (None, 12) │ 6,156 │ add\_1[0][0] │**

**└─────────────────────┴───────────────────┴────────────**

**┴───────────────────┘ Total params: 14,429,708 (55.04 MB)**

**Trainable params: 14,429,708 (55.04 MB)**

**Non-trainable params: 0 (0.00 B)**

1. **Generating a caption for the sample image (using dummy model output)...**

**Predicted Caption: "the the the the the the the the the the the the the the the"**

****

**--- End of Demonstration ---**

**Remember to train a real model with a proper dataset for meaningful captions.**

# PRACT 6

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

## CODE :

import tensorflow as tf

from tensorflow.keras import layers, models

from sklearn.model\_selection import train\_test\_split from sklearn.preprocessing import MinMaxScaler import numpy as np

import matplotlib.pyplot as plt

# 1. Load and Preprocess Real-World Data (Example: Boston Housing Dataset) # Replace this with your actual data loading and preprocessing steps.

def load\_and\_preprocess\_data():

from sklearn.datasets import load\_boston boston = load\_boston()

data = boston.data

target = boston.target # We won't use the target for autoencoding, but it's here. #print(boston.DESCR) #Uncomment to see description

# Split data

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

# Scale the data to the range [0, 1] using MinMaxScaler scaler = MinMaxScaler()

X\_train\_scaled = scaler.fit\_transform(X\_train) X\_test\_scaled = scaler.transform(X\_test)

return X\_train\_scaled, X\_test\_scaled, scaler # Return the scaler as well

# 2. Define the Autoencoder Model

def create\_autoencoder(input\_dim, encoding\_dim): """

Creates a simple autoencoder model.

Args:

input\_dim: The dimension of the input data.

encoding\_dim: The dimension of the encoded representation.

Returns:

A Keras model representing the autoencoder. """

# Encoder

input\_layer = layers.Input(shape=(input\_dim,))

encoder = layers.Dense(128, activation='relu')(input\_layer) # Example layer encoder = layers.Dense(64, activation='relu')(encoder) # Example layer

encoding\_layer = layers.Dense(encoding\_dim, activation='relu')(encoder) #changed from 'linear'

# Decoder

decoder = layers.Dense(64, activation='relu')(encoding\_layer) # Example layer decoder = layers.Dense(128, activation='relu')(decoder) # Example layer

output\_layer = layers.Dense(input\_dim, activation='linear')(decoder) # Use 'linear' for regression

autoencoder = models.Model(inputs=input\_layer, outputs=output\_layer) autoencoder.compile(optimizer='adam', loss='mse') # Use 'mse' for real-valued data return autoencoder

# 3. Train the Autoencoder

def train\_autoencoder(autoencoder, X\_train, X\_test, epochs=100, batch\_size=32): """

Trains the autoencoder model.

Args:

autoencoder: The Keras autoencoder model. X\_train: The training data.

X\_test: The testing data.

epochs: Number of training epochs. batch\_size: The batch size.

Returns:

The training history object. """

history = autoencoder.fit(X\_train, X\_train, # Autoencoders reconstruct the input epochs=epochs,

batch\_size=batch\_size, shuffle=True, validation\_data=(X\_test, X\_test), verbose=0) #Added verbose=0

return history

# 4. Encode and Decode Data

def encode\_and\_decode(autoencoder, data): """

Encodes and decodes data using the trained autoencoder.

Args:

autoencoder: The trained Keras autoencoder model. data: The data to encode and decode.

Returns:

The encoded and decoded data. """

encoder\_model = models.Model(inputs=autoencoder.input, outputs=autoencoder.layers[2].output) #changed index

encoded\_data = encoder\_model.predict(data, verbose=0) decoded\_data = autoencoder.predict(data, verbose=0) return encoded\_data, decoded\_data

# 5. Evaluate Results and Visualize (Optional)

def evaluate\_and\_visualize(history, X\_test, decoded\_data): """

Evaluates the autoencoder's performance and visualizes the results.

Args:

history: The training history object. X\_test: The original test data decoded\_data: The decoded test data.

"""

plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)

plt.plot(history.history['loss'], label='Training Loss') plt.plot(history.history['val\_loss'], label='Validation Loss') plt.title('Loss Curve')

plt.xlabel('Epoch') plt.ylabel('Loss') plt.legend()

plt.subplot(1, 2, 2)

#print(X\_test.shape, decoded\_data.shape)

num\_samples = min(10, len(X\_test)) # Limit the number of samples to visualize np.random.seed(42)

indices = np.random.choice(len(X\_test), num\_samples, replace=False) #select random indices for i, index in enumerate(indices):

plt.plot(X\_test[index], label=f'Original {i+1}', linestyle='--') plt.plot(decoded\_data[index], label=f'Decoded {i+1}')

plt.title('Original vs. Decoded Data') plt.xlabel('Feature') plt.ylabel('Value')

plt.legend() plt.tight\_layout() plt.show()

def main():

# 1. Load and Preprocess Data

X\_train\_scaled, X\_test\_scaled, scaler = load\_and\_preprocess\_data() input\_dim = X\_train\_scaled.shape[1] # Get the number of features encoding\_dim = 8 # Choose the dimension of the encoded representation

# 2. Create Autoencoder Model

autoencoder = create\_autoencoder(input\_dim, encoding\_dim) autoencoder.summary()

# 3. Train Autoencoder

history = train\_autoencoder(autoencoder, X\_train\_scaled, X\_test\_scaled, epochs=100, batch\_size=32)

# 4. Encode and Decode Data

encoded\_data, decoded\_data = encode\_and\_decode(autoencoder, X\_test\_scaled) print("Encoded Data Shape:", encoded\_data.shape)

print("Decoded Data Shape:", decoded\_data.shape)

# 5. Evaluate and Visualize Results evaluate\_and\_visualize(history, X\_test\_scaled, decoded\_data)

# 6. Using the Encoder for Feature Extraction

# You can now use the 'encoded\_data' as a lower-dimensional representation of your original data.

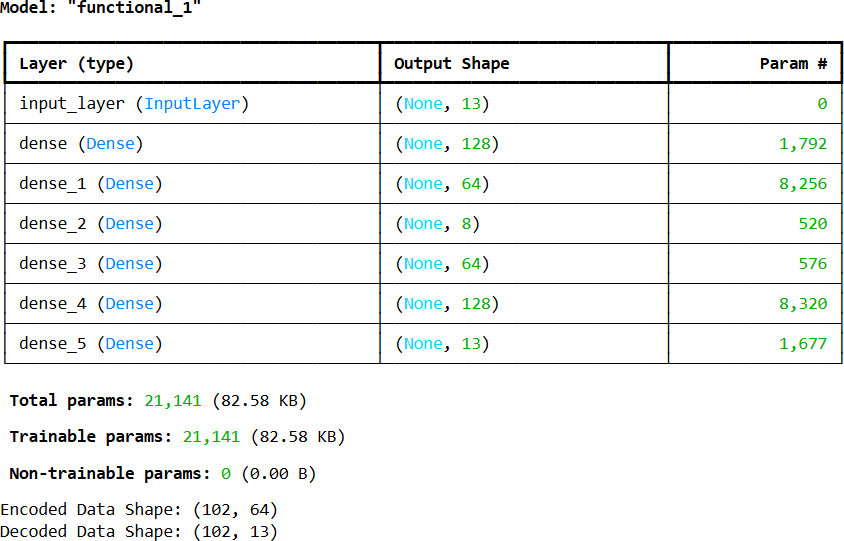
# This can be useful for visualization, clustering, or as input to another machine learning model.

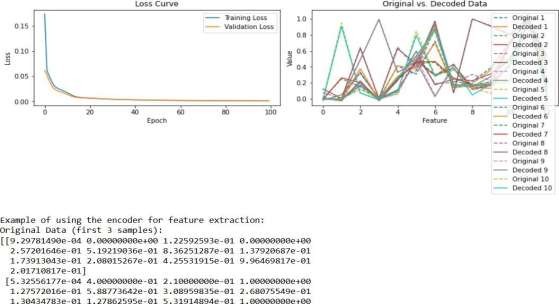
print("\nExample of using the encoder for feature extraction:") print("Original Data (first 3 samples):") print(X\_test\_scaled[:3])

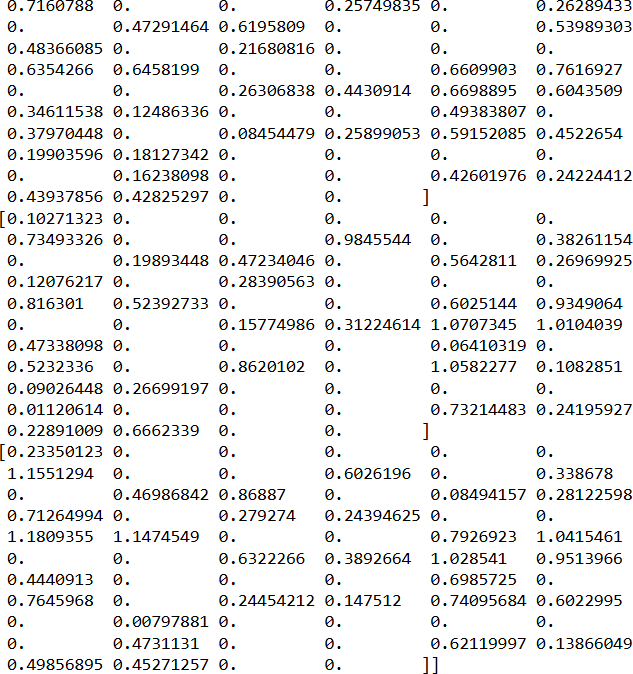
print("Encoded Data (first 3 samples):") print(encoded\_data[:3])

if name == " main ": main()

## OUTPUT:

****



****

**PRACT 6B**:

import numpy as np import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

# 1. Generate some simple real-world-like data (replace with your actual data) # Let's say we have sensor readings with 5 features

np.random.seed(42) input\_dim = 5

num\_samples = 1000

real\_world\_data = np.random.rand(num\_samples, input\_dim)

# Normalize the data (important for neural networks) max\_vals = np.max(real\_world\_data, axis=0) min\_vals = np.min(real\_world\_data, axis=0)

normalized\_data = (real\_world\_data - min\_vals) / (max\_vals - min\_vals + 1e-8) # Adding a small epsilon to avoid division by zero

# Split data into training and testing sets train\_ratio = 0.8

train\_size = int(train\_ratio \* num\_samples) train\_data = normalized\_data[:train\_size] test\_data = normalized\_data[train\_size:]

# 2. Define the Autoencoder Model

# We'll create a simple autoencoder with one encoder and one decoder layer

# Encoder

encoding\_dim = 2 # The dimensionality of the encoded representation (bottleneck) encoder\_input = keras.Input(shape=(input\_dim,))

encoded = layers.Dense(encoding\_dim, activation='relu')(encoder\_input)

# Decoder

decoded = layers.Dense(input\_dim, activation='sigmoid')(encoded) # Using sigmoid for normalized data (0 to 1)

# Autoencoder model

autoencoder = keras.Model(encoder\_input, decoded)

# Encoder model (to get the encoded representation) encoder = keras.Model(encoder\_input, encoded)

# Decoder model (to decode an encoded representation) encoded\_input = keras.Input(shape=(encoding\_dim,)) decoder\_layer = autoencoder.layers[-1]

decoder = keras.Model(encoded\_input, decoder\_layer(encoded\_input))

# 3. Compile the Autoencoder

autoencoder.compile(optimizer='adam', loss='mse') # Mean Squared Error is common for reconstruction tasks

# 4. Train the Autoencoder epochs = 50

batch\_size = 32

history = autoencoder.fit(train\_data, train\_data, epochs=epochs, batch\_size=batch\_size, shuffle=True,

validation\_data=(test\_data, test\_data))

# 5. Encode the real-world data

encoded\_data = encoder.predict(normalized\_data) print("\nEncoded Data (first 5 samples):\n", encoded\_data[:5])

# 6. Decode the encoded data (to see the reconstruction) decoded\_data = decoder.predict(encoded\_data) print("\nDecoded Data (first 5 samples):\n", decoded\_data[:5])

# 7. Original Data (first 5 samples) for comparison

print("\nOriginal Normalized Data (first 5 samples):\n", normalized\_data[:5])

# 8. Evaluate the Autoencoder (reconstruction error)

loss = autoencoder.evaluate(test\_data, test\_data, verbose=0) print(f"\nTest Loss (Mean Squared Error): {loss:.4f}")

# Optional: Visualize the training history import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training Loss') plt.plot(history.history['val\_loss'], label='Validation Loss') plt.title('Autoencoder Training History') plt.xlabel('Epoch')

plt.ylabel('Loss') plt.legend() plt.show()

# Optional: If the encoding dimension is 2, you can visualize the encoded data if encoding\_dim == 2:

plt.figure(figsize=(8, 6))

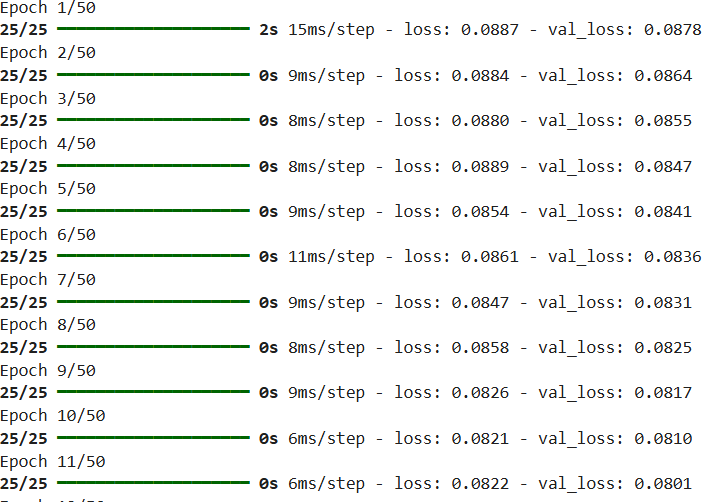
plt.scatter(encoded\_data[:, 0], encoded\_data[:, 1], c='blue', alpha=0.5)

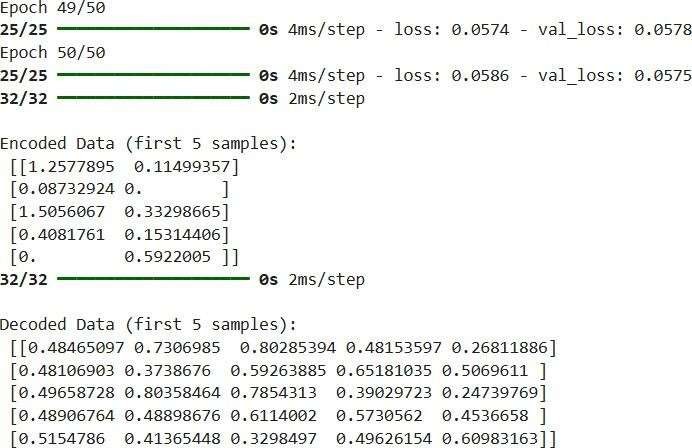
plt.title('Encoded Data in 2D') plt.xlabel('Encoded Feature 1')

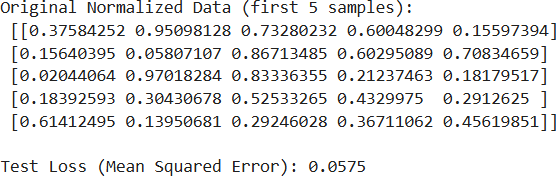
plt.ylabel('Encoded Feature 2') plt.grid(True)

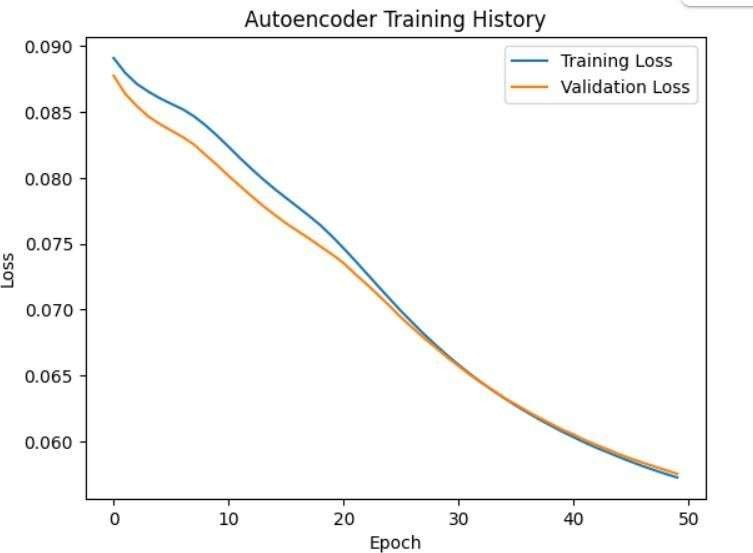
plt.show()

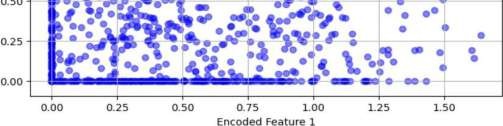
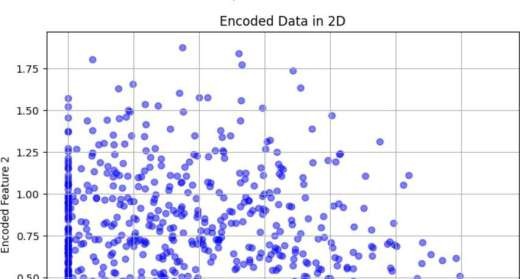
**OUTPUT:**

****



****





# PRACTICAL 7

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

import numpy as np import tensorflow as tf

from tensorflow.keras.models import Sequential

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

Dropout

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.datasets import mnist # You can replace this with other character datasets import matplotlib.pyplot as plt

# 1. Load and Preprocess the Data

# Load the MNIST dataset (for simplicity). You can replace this with your own character dataset.

# If you replace it, ensure your images are grayscale and relatively small. (x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Preprocess the data img\_width, img\_height = 28, 28

num\_classes = 10 # MNIST has 10 digits (0-9)

# Normalize pixel values to be between 0 and 1 x\_train = x\_train.astype('float32') / 255.0 x\_test = x\_test.astype('float32') / 255.0

# One-hot encode the labels

y\_train = to\_categorical(y\_train, num\_classes) y\_test = to\_categorical(y\_test, num\_classes)

# 2. Define the Models

# 2.1 RNN Model for Character Recognition

def create\_rnn\_model(img\_width, img\_height, num\_classes): """

Creates an RNN model for character recognition. The input images are reshaped to sequences, and an LSTM network is used to classify them.

Args:

img\_width: The width of the input images. img\_height: The height of the input images.

num\_classes: The number of classes (characters) to recognize.

Returns:

A Keras Sequential model. """

model = Sequential()

# Reshape the input images to sequences of pixel values. The LSTM will process # each row of the image as a sequence.

model.add(Reshape((img\_width, img\_height), input\_shape=(img\_width, img\_height))) # Add an LSTM layer. We're using a relatively small number of units here. model.add(LSTM(128, return\_sequences=False)) # You can experiment with more units model.add(Dropout(0.2))

# Add a dense layer for classification model.add(Dense(num\_classes, activation='softmax')) return model

# 2.2 CNN Model for Character Recognition

def create\_cnn\_model(img\_width, img\_height, num\_classes):

model = Sequential()

# Convolutional layers to extract features. We use small filters (3x3) and # max pooling to reduce the spatial dimensions.

model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=(img\_width, img\_height, 1)))

model.add(MaxPooling2D((2, 2)))

model.add(Conv2D(64, (3, 3), activation='relu'))

model.add(MaxPooling2D((2, 2)))

# Flatten the feature maps before feeding them into a dense layer model.add(Flatten())

model.add(Dropout(0.2))

# Dense layers for classification model.add(Dense(128, activation='relu')) model.add(Dense(num\_classes, activation='softmax')) return model

# Create the models

rnn\_model = create\_rnn\_model(img\_width, img\_height, num\_classes) cnn\_model = create\_cnn\_model(img\_width, img\_height, num\_classes)

# Print model summaries print("RNN Model Summary:") rnn\_model.summary() print("\nCNN Model Summary:") cnn\_model.summary()

# 3. Train the Models

# Reshape the training and testing data for the RNN model x\_train\_rnn = x\_train.reshape(-1, img\_width, img\_height) x\_test\_rnn = x\_test.reshape(-1, img\_width, img\_height)

# Add a channel dimension to the CNN input

x\_train\_cnn = x\_train.reshape(-1, img\_width, img\_height, 1) x\_test\_cnn = x\_test.reshape(-1, img\_width, img\_height, 1)

# Compile the models

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

# Train the models

epochs = 10 # You can adjust this batch\_size = 128

print("\nTraining RNN Model:")

rnn\_history = rnn\_model.fit(x\_train\_rnn, y\_train, epochs=epochs, batch\_size=batch\_size, validation\_data=(x\_test\_rnn, y\_test), verbose=0)

print("\nTraining CNN Model:")

cnn\_history = cnn\_model.fit(x\_train\_cnn, y\_train, epochs=epochs, batch\_size=batch\_size, validation\_data=(x\_test\_cnn, y\_test), verbose=0)

# 4. Evaluate the Models

# Evaluate the models on the test set print("\nEvaluating RNN Model:")

rnn\_loss, rnn\_accuracy = rnn\_model.evaluate(x\_test\_rnn, y\_test, verbose=0) print(f"RNN Test Loss: {rnn\_loss:.4f}, RNN Test Accuracy: {rnn\_accuracy:.4f}")

print("\nEvaluating CNN Model:")

cnn\_loss, cnn\_accuracy = cnn\_model.evaluate(x\_test\_cnn, y\_test, verbose=0) print(f"CNN Test Loss: {cnn\_loss:.4f}, CNN Test Accuracy: {cnn\_accuracy:.4f}")

# 5. Visualize the Results

# Plot the training and validation accuracy for both models plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)

plt.plot(rnn\_history.history['accuracy'], label='RNN Train Accuracy') plt.plot(rnn\_history.history['val\_accuracy'], label='RNN Val Accuracy') plt.title('RNN Accuracy')

plt.xlabel('Epoch') plt.ylabel('Accuracy') plt.legend()

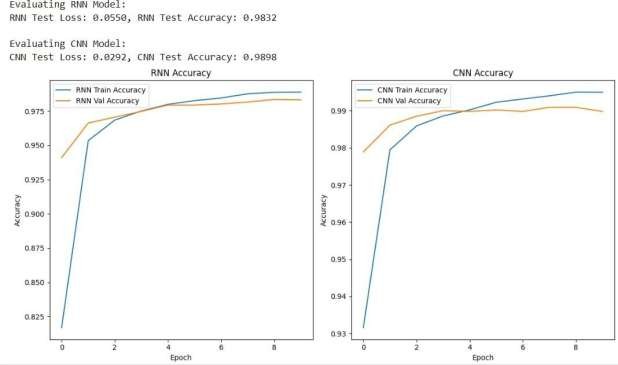
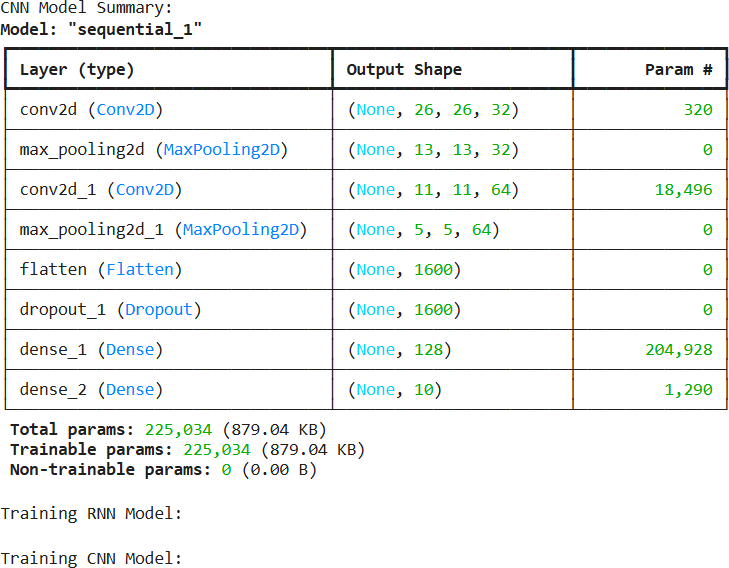
plt.subplot(1, 2, 2)

plt.plot(cnn\_history.history['accuracy'], label='CNN Train Accuracy') plt.plot(cnn\_history.history['val\_accuracy'], label='CNN Val Accuracy') plt.title('CNN Accuracy')

plt.xlabel('Epoch') plt.ylabel('Accuracy') plt.legend()

plt.tight\_layout() plt.show()

**OUTPUT:**

****

# PRACTICAL N0. 8

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

## CODE:

import tensorflow as tf

from tensorflow.keras.layers import Input, Dense from tensorflow.keras.models import Model import numpy as np

import matplotlib.pyplot as plt

# --- 1. Load and Preprocess the MNIST Dataset --- print("1. Loading and preprocessing the MNIST dataset...") (x\_train, \_), (x\_test, \_) = tf.keras.datasets.mnist.load\_data()

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

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:]))) x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

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

# --- 2. Define the Autoencoder Model ---

print("\n2. Defining the Autoencoder model architecture...") encoding\_dim = 32

input\_img = Input(shape=(784,)) # Hidden layer for the encoder

encoded = Dense(128, activation='relu')(input\_img) encoded = Dense(encoding\_dim, activation='relu')(encoded) decoded = Dense(128, activation='relu')(encoded)

decoded = Dense(784, activation='sigmoid')(decoded) # Output of decoder autoencoder = Model(inputs=input\_img, outputs=decoded)

encoder = Model(inputs=input\_img, outputs=encoded) encoded\_input = Input(shape=(encoding\_dim,)) decoder\_layer\_1 = autoencoder.layers[-2] decoder\_layer\_2 = autoencoder.layers[-1]

decoder =Model(inputs=encoded\_input, outputs=decoder\_layer\_2(decoder\_layer\_1(encoded\_input)))

# --- 3. Compile the Autoencoder Model ---

print("\n3. Compiling the Autoencoder model...") autoencoder.compile(optimizer='adam', loss='binary\_crossentropy') autoencoder.summary()

# --- 4. Train the Autoencoder Model ---

print("\n4. Training the Autoencoder model...") history = autoencoder.fit(x\_train, x\_train,

epochs=50, # batch\_size=256, shuffle=True,

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

# --- 5. Visualize Training History (Optional) ---

print("\n5. Plotting training loss history...") plt.figure(figsize=(10, 5)) plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss') plt.title('Autoencoder Training Loss')

plt.xlabel('Epoch') plt.ylabel('Loss') plt.legend() plt.grid(True) plt.show()

# --- 6. Visualize Original vs. Reconstructed Digits ---

print("\n6. Visualizing original and reconstructed digits...") num\_images\_to\_display = 10

encoded\_imgs = encoder.predict(x\_test[:num\_images\_to\_display]) reconstructed\_imgs = decoder.predict(encoded\_imgs) plt.figure(figsize=(20, 4))

for i in range(num\_images\_to\_display):

ax = plt.subplot(2, num\_images\_to\_display, i + 1) plt.imshow(x\_test[i].reshape(28, 28))

plt.gray() ax.get\_xaxis().set\_visible(False) ax.get\_yaxis().set\_visible(False) plt.title("Original")

# Reconstructed Image

ax = plt.subplot(2, num\_images\_to\_display, i + 1 + num\_images\_to\_display) plt.imshow(reconstructed\_imgs[i].reshape(28, 28)) # Reshape back for display plt.gray()

ax.get\_xaxis().set\_visible(False) ax.get\_yaxis().set\_visible(False) plt.title("Reconstructed")

plt.suptitle("Original vs. Reconstructed MNIST Digits") plt.show()

print("\n--- Autoencoder Demonstration Complete ---")

print("The plots above show the training progress and the quality of reconstruction.")

## OUTPUT:

1. Loading and preprocessing the MNIST dataset...

Downloading data from <https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>

**11490434/11490434** ━━━━━━━━━━━━━━━━━━━━ **0s** 0us/step Training data shape: (60000, 784)

Test data shape: (10000, 784)

1. Defining the Autoencoder model architecture...
2. Compiling the Autoencoder model...

**Model: "functional"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━

━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━

━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩

│ input\_layer (InputLayer) │ (None, 784) │ 0 │

├─────────────────────────────────┼────────────────────

────┼───────────────┤

│ dense (Dense) │ (None, 128) │ 100,480 │

├─────────────────────────────────┼────────────────────

────┼───────────────┤

│ dense\_1 (Dense) │ (None, 32) │ 4,128 │

├─────────────────────────────────┼────────────────────

────┼───────────────┤

│ dense\_2 (Dense) │ (None, 128) │ 4,224 │

├─────────────────────────────────┼────────────────────

────┼───────────────┤

│ dense\_3 (Dense) │ (None, 784) │ 101,136 │

└─────────────────────────────────┴────────────────────

────┴───────────────┘

**Total params:** 209,968 (820.19 KB)

**Trainable params:** 209,968 (820.19 KB)

**Non-trainable params:** 0 (0.00 B)

1. Training the Autoencoder model... Epoch 1/50

**235/235** ━━━━━━━━━━━━━━━━━━━━ **9s** 26ms/step - loss: 0.3234 - val\_loss: 0.1506

Epoch 2/50

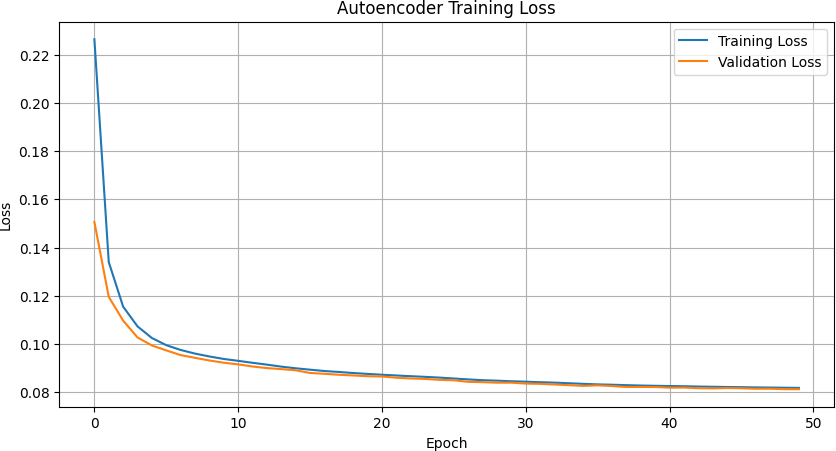
**235/235** ━━━━━━━━━━━━━━━━━━━━ **4s** 16ms/step - loss: 0.1417 - val\_loss: 0.1196

Epoch 3/50

**235/235** ━━━━━━━━━━━━━━━━━━━━ **4s** 16ms/step - loss: 0.1179 - val\_loss: 0.1097

Epoch 4/50

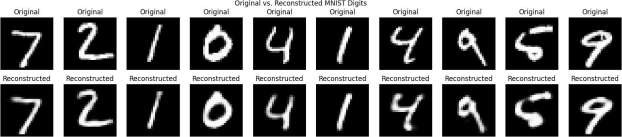
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **235/235** ━━━━━━━━━━━━━━━━━━━━  val\_loss: 0.1027  Epoch 5/50 | **6s** | 19ms/step | - loss: | 0.1089 | - |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **4s** | 16ms/step | - loss: | 0.1034 | - |
| val\_loss: 0.0994  Epoch 6/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━  val\_loss: 0.0974  Epoch 7/50 | **4s** | 16ms/step | - loss: | 0.1002 | - |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **6s** | 18ms/step | - loss: | 0.0979 | - |
| val\_loss: 0.0954  Epoch 8/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **5s** | 16ms/step | - loss: | 0.0963 | - |
| val\_loss: 0.0943  …. |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **4s** | 16ms/step | - loss: | 0.0822 | - |
| val\_loss: 0.0818  Epoch 46/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━  val\_loss: 0.0817  Epoch 47/50 | **5s** | 17ms/step | - loss: | 0.0822 | - |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **5s** | 18ms/step | - loss: | 0.0819 | - |
| val\_loss: 0.0815  Epoch 48/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **5s** | 16ms/step | - loss: | 0.0820 | - |
| val\_loss: 0.0815  Epoch 49/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **5s** | 21ms/step | - loss: | 0.0819 | - |
| val\_loss: 0.0813  Epoch 50/50 |  |  |  |  |  |
| **235/235** ━━━━━━━━━━━━━━━━━━━━ | **4s** | 16ms/step | - loss: | 0.0817 | - |
| val\_loss: 0.0813 |  |  |  |  |  |
| 5. Plotting training loss history... |  |  |  |  |  |



6. Visualizing original and reconstructed digits...

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 68ms/step

**1/1** ━━━━━━━━━━━━━━━━━━━━ **0s** 67ms/step



--- Autoencoder Demonstration Complete ---

The plots above show the training progress and the quality of reconstruction

# PRACTICAL N0.9

**Demonstrate recurrent neural network that learns to perform sequence analysis for stock**

**price.(google stock price)**

## 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 LSTM, Dense from sklearn.preprocessing import MinMaxScaler from sklearn.metrics import mean\_squared\_error

# --- 1. Generate Simulated Google Stock Price Data --- print("1. Generating simulated Google stock price data...")

# Number of data points num\_data\_points = 1000

time = np.arange(0, num\_data\_points) # Base trend

trend = time \* 0.1

seasonality = 10 \* np.sin(time / 50) + 5 \* np.cos(time / 20) # Random noise

noise = np.random.normal(loc=0, scale=2, size=num\_data\_points) simulated\_stock\_price = np.maximum(0, 50 + trend + seasonality + noise) simulated\_stock\_price = simulated\_stock\_price.reshape(-1, 1)

print(f"Simulated stock price data shape: {simulated\_stock\_price.shape}") # --- 2. Data Preprocessing ---

print("\n2. Preprocessing data: Normalization and sequence creation...") scaler = MinMaxScaler(feature\_range=(0, 1))

scaled\_data = scaler.fit\_transform(simulated\_stock\_price) train\_size = int(len(scaled\_data) \* 0.8)

train\_data, test\_data = scaled\_data[0:train\_size,:], scaled\_data[train\_size:len(scaled\_data),:] def create\_dataset(dataset, look\_back=1):

X, Y = [], []

for i in range(len(dataset) - look\_back - 1): a = dataset[i:(i + look\_back), 0] X.append(a)

Y.append(dataset[i + look\_back, 0]) return np.array(X), np.array(Y)

look\_back = 10

X\_train, y\_train = create\_dataset(train\_data, look\_back) X\_test, y\_test = create\_dataset(test\_data, look\_back)

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

print(f"X\_train shape: {X\_train.shape}") print(f"y\_train shape: {y\_train.shape}")

print(f"X\_test shape: {X\_test.shape}") print(f"y\_test shape: {y\_test.shape}")

# --- 3. Build the LSTM Model ---

print("\n3. Building the LSTM model architecture...") model = Sequential()

model.add(LSTM(50, input\_shape=(look\_back, 1))) model.add(Dense(1))

# --- 4. Compile and Train the Model --- print("\n4. Compiling and training the model...")

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

# Print model summary model.summary()

# Train the model

history = model.fit(X\_train, y\_train,

epochs=100, # Number of training iterations (can be increased) batch\_size=32,

verbose=1, # Show training progress validation\_data=(X\_test, y\_test))

# --- 5. Make Predictions ---

print("\n5. Making predictions on training and test data...")

# Make predictions on the training and test sets train\_predict = model.predict(X\_train) test\_predict = model.predict(X\_test)

train\_predict = scaler.inverse\_transform(train\_predict)

y\_train\_orig = scaler.inverse\_transform(y\_train.reshape(-1, 1)) # Reshape for inverse\_transform test\_predict = scaler.inverse\_transform(test\_predict)

y\_test\_orig = scaler.inverse\_transform(y\_test.reshape(-1, 1)) # Reshape for inverse\_transform train\_rmse = np.sqrt(mean\_squared\_error(y\_train\_orig, train\_predict))

test\_rmse = np.sqrt(mean\_squared\_error(y\_test\_orig, test\_predict)) print(f"Train RMSE: {train\_rmse:.2f}")

print(f"Test RMSE: {test\_rmse:.2f}")

# --- 6. Visualize Results ---

print("\n6. Visualizing original, training predictions, and test predictions...") # Shift train predictions for plotting

train\_predict\_plot = np.empty\_like(simulated\_stock\_price)

train\_predict\_plot[:, :] = np.nan train\_predict\_plot[look\_back:len(train\_predict)+look\_back, :] = train\_predict

# Shift test predictions for plotting

test\_predict\_plot = np.empty\_like(simulated\_stock\_price) test\_predict\_plot[:, :] = np.nan

test\_predict\_plot[len(train\_predict)+(look\_back\*2)+1:len(simulated\_stock\_price)-1, :] = test\_predict

# Plot baseline and predictions plt.figure(figsize=(15, 7))

plt.plot(simulated\_stock\_price, label='Original Simulated Stock Price', color='blue') plt.plot(train\_predict\_plot, label='Training Predictions', color='green', linestyle='--') plt.plot(test\_predict\_plot, label='Test Predictions', color='red', linestyle='--') plt.title('Simulated Google Stock Price Prediction using LSTM')

plt.xlabel('Time Step') plt.ylabel('Stock Price') plt.legend() plt.grid(True) plt.show()

# Plot training and validation loss plt.figure(figsize=(10, 5)) plt.plot(history.history['loss'], label='Training Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss') plt.title('Model Loss During Training')

plt.xlabel('Epoch') plt.ylabel('Loss') plt.legend()

plt.grid(True) plt.show()

print("\n--- RNN for Stock Price Prediction Demonstration Complete ---")

print("The plots show how the LSTM model attempts to learn the patterns in the simulated stock data.")

print("For real stock data, factors like news, economic indicators, and market sentiment would also play a role.")

## OUTPUT:

1. Generating simulated Google stock price data... Simulated stock price data shape: (1000, 1)
2. Preprocessing data: Normalization and sequence creation... X\_train shape: (789, 10, 1)

y\_train shape: (789,)

X\_test shape: (189, 10, 1)

y\_test shape: (189,)

1. Building the LSTM model architecture...

/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super(). init (\*\*kwargs)

1. Compiling and training the model...

**Model: "sequential"**

┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━

━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓

┃ **Layer (type)** ┃ **Output Shape** ┃ **Param #** ┃

┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━

━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩

│ lstm (LSTM) │ (None, 50) │ 10,400 │

├─────────────────────────────────┼────────────────────

────┼───────────────┤

│ dense (Dense) │ (None, 1) │ 51 │

└─────────────────────────────────┴────────────────────

────┴───────────────┘

**Total params:** 10,451 (40.82 KB)

**Trainable params:** 10,451 (40.82 KB)

**Non-trainable params:** 0 (0.00 B) Epoch 1/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **11s** 21ms/step - loss: 0.0825 - val\_loss: 0.0014

Epoch 2/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 11ms/step - loss: 0.0046 - val\_loss: 0.0018

Epoch 3/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 0.0012 - val\_loss: 0.0010

Epoch 4/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 6.1705e-04 - val\_loss: 4.3283e-04

Epoch 5/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 5.0940e-04 - val\_loss: 4.0927e-04

Epoch 6/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - loss: 5.5506e-04 - val\_loss: 4.0690e-04

Epoch 7/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - loss: 4.9188e-04 - val\_loss: 4.1111e-04

Epoch 8/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 4.8884e-04 - val\_loss: 4.1891e-04

Epoch 9/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - loss: 4.8215e-04 - val\_loss: 4.2358e-04

Epoch 10/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 4.8328e-04 - val\_loss: 4.3879e-04

Epoch 11/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - loss: 4.4245e-04 - val\_loss: 7.2078e-04

Epoch 99/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 9ms/step - loss: 4.6183e-04 - val\_loss: 0.0011

Epoch 100/100

**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step - loss: 4.8694e-04 -

val\_loss: 7.0814e-04

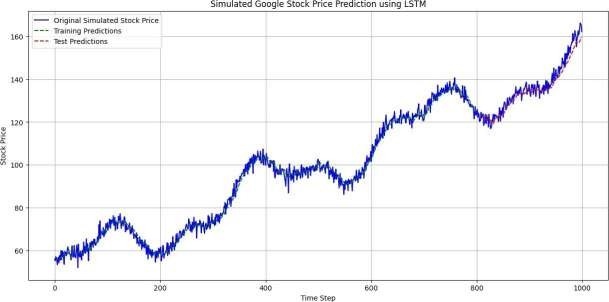
1. Making predictions on training and test data...

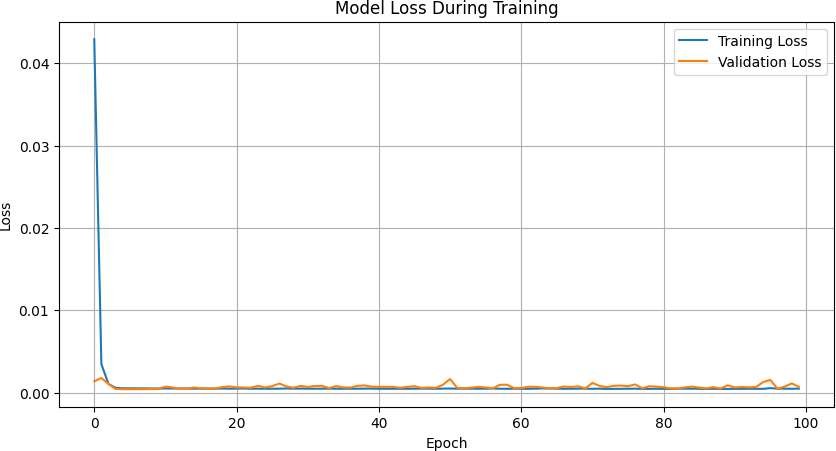
**25/25** ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step

**6/6** ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step Train RMSE: 2.40

Test RMSE: 3.04

1. Visualizing original, training predictions, and test predictions...





--- RNN for Stock Price Prediction Demonstration Complete ---

The plots show how the LSTM model attempts to learn the patterns in the simulated stock data. For real stock data, factors like news, economic indicators, and market sentiment would also play a role.

# PRACTICAL NO.10

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

## CODE:

import numpy as np import tensorflow as tf

from tensorflow.keras import layers, models, optimizers import matplotlib.pyplot as plt

import os

# --- 1. Define Generator Model ---

# The Generator takes random noise as input and tries to produce realistic images. def model = models.Sequential()

model.add(layers.Dense(7 \* 7 \* 256, use\_bias=False, input\_shape=(latent\_dim,))) model.add(layers.BatchNormalization()) # Stabilizes training model.add(layers.LeakyReLU()) # Non-linear activation

# Reshape to 7x7x256, which is the input shape for the first Conv2DTranspose layer. model.add(layers.Reshape((7, 7, 256)))

assert model.output\_shape == (None, 7, 7, 256) # None is batch size

# First upsampling block: Upsample to 14x14

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

model.add(layers.BatchNormalization()) model.add(layers.LeakyReLU())

assert model.output\_shape == (None, 7, 7, 128)

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

model.add(layers.BatchNormalization()) model.add(layers.LeakyReLU())

assert model.output\_shape == (None, 14, 14, 64)

# Second upsampling block: Upsample to 28x28

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, 28, 28, 1) return model

# --- 2. Define Discriminator Model ---

# The Discriminator takes an image as input and tries to classify it as real or fake. def model = models.Sequential()

# First convolutional block

model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input\_shape=image\_shape))

model.add(layers.LeakyReLU()) model.add(layers.Dropout(0.3)) # Prevents overfitting

# Second convolutional block

model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same')) model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

# Flatten the output for the Dense layer model.add(layers.Flatten())

# Output layer: Single neuron with sigmoid activation for binary classification (real/fake) model.add(layers.Dense(1, activation='sigmoid'))

return model

# --- 3. Define the GAN (Combined Model) ---

# The GAN combines the Generator and Discriminator for training. def build\_gan(generator, discriminator):

# The discriminator is not trainable when training the generator discriminator.trainable = False

# Input to the GAN is the noise vector gan\_input = layers.Input(shape=(latent\_dim,)) # Generate an image from the noise generated\_image = generator(gan\_input)

# The discriminator tries to classify the generated image gan\_output = discriminator(generated\_image)

# Create the combined model

gan = models.Model(gan\_input, gan\_output) return gan

# --- 4. Training Function ---

def train\_gan(generator, discriminator, gan, dataset, latent\_dim, epochs=50, batch\_size=128): # Define optimizers and loss functions

d\_optimizer = optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5) g\_optimizer = optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5)

# Compile the discriminator

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

# Compile the combined GAN (only generator is trained here) gan.compile(loss='binary\_crossentropy', optimizer=g\_optimizer)

# For plotting generated images during training

seed = tf.random.normal([16, latent\_dim]) # Generate 16 images for visualization

# Training loop

for epoch in range(epochs): print(f"Epoch {epoch + 1}/{epochs}")

for i, real\_images in enumerate(dataset): #

# Train Discriminator #

# Generate random noise

noise = tf.random.normal([batch\_size, latent\_dim]) # Generate fake images

fake\_images = generator(noise, training=True)

# Combine real and fake images for discriminator training combined\_images = tf.concat([real\_images, fake\_images], axis=0)

# Create labels: 1 for real, 0 for fake

labels = tf.concat([tf.ones((real\_images.shape[0], 1)), tf.zeros((fake\_images.shape[0], 1))], axis=0)

# Add some random noise to the labels (label smoothing) to help stabilize training labels += 0.05 \* tf.random.uniform(labels.shape)

# Train the discriminator

d\_loss, d\_accuracy = discriminator.train\_on\_batch(combined\_images, labels)

#

# Train Generator #

# Generate random noise for generator training noise = tf.random.normal([batch\_size, latent\_dim])

# Generator wants discriminator to classify generated images as real (label 1) misleading\_labels = tf.ones((batch\_size, 1))

# Train the generator (via the combined GAN model) g\_loss = gan.train\_on\_batch(noise, misleading\_labels)

if i % 100 == 0:

print(f" Batch {i}: D\_loss={d\_loss:.4f}, D\_acc={d\_accuracy:.4f}, G\_loss={g\_loss:.4f}")

# Save and plot generated images at the end of each epoch generate\_and\_save\_images(generator, epoch + 1, seed)

# After training, generate a final set of images generate\_and\_save\_images(generator, epochs, seed, final=True)

# --- 5. Image Generation and Plotting Function ---

def generate\_and\_save\_images(model, epoch, test\_input, final=False): predictions = model(test\_input, training=False)

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

for i in range(predictions.shape[0]): plt.subplot(4, 4, i+1)

# Rescale images from tanh output (-1 to 1) to (0 to 1)

plt.imshow((predictions[i, :, :, 0] + 1) / 2, cmap='gray') plt.axis('off')

if not final:

plt.suptitle(f'Epoch {epoch}', y=1.02) plt.savefig(f'gan\_image\_at\_epoch\_{epoch:04d}.png')

else:

plt.suptitle(f'Final Generated Images (Epoch {epoch})', y=1.02) plt.savefig(f'gan\_final\_images.png')

plt.show() plt.close(fig)

# --- Main Execution Block --- if name == ' main ':

print("--- Starting GAN Image Generation Example ---")

# Parameters

latent\_dim = 100 # Dimension of the noise vector

epochs = 10 # Number of training epochs (increase for better results) batch\_size = 128

image\_shape = (28, 28, 1) # For MNIST-like images

# Load and preprocess the dataset (using MNIST for simplicity) print("Loading MNIST dataset...")

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

# Normalize images to [-1, 1] range (common for GANs with tanh activation in generator) x\_train = x\_train.reshape(x\_train.shape[0], \*image\_shape).astype('float32')

x\_train = (x\_train - 127.5) / 127.5 # Normalize to [-1, 1]

# Create a tf.data.Dataset for efficient loading

train\_dataset = tf.data.Dataset.from\_tensor\_slices(x\_train).shuffle(60000).batch(batch\_size) print(f"MNIST dataset loaded. Number of training samples: {x\_train.shape[0]}")

# Build the models

print("Building Generator, Discriminator, and GAN models...") generator = build\_generator(latent\_dim)

discriminator = build\_discriminator(image\_shape) gan = build\_gan(generator, discriminator) print("Models built.")

print("Starting GAN training...")

train\_gan(generator, discriminator, gan, train\_dataset, latent\_dim, epochs, batch\_size) print("GAN training finished.")

print("--- GAN Image Generation Example Finished ---")

## OUTPUT:

--- Starting GAN Image Generation Example --- Loading MNIST dataset...

Downloading data from<https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz>

**11490434/11490434** ━━━━━━━━━━━━━━━━━━━━ **0s** 0us/step MNIST dataset loaded. Number of training samples: 60000

Building Generator, Discriminator, and GAN models...

/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super(). init (activity\_regularizer=activity\_regularizer, \*\*kwargs)

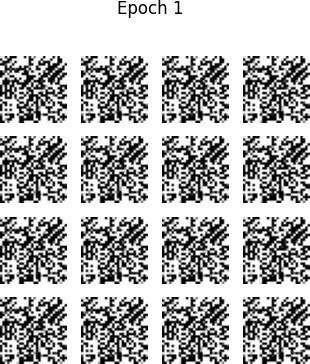
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead. super().init(activity\_regularizer=activity\_regularizer, \*\*kwargs)

Models built.

Starting GAN training... Epoch 1/10

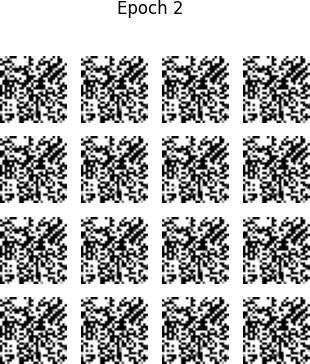
/usr/local/lib/python3.11/dist-packages/keras/src/backend/tensorflow/trainer.py:82: UserWarning: The model does not have any trainable weights. warnings.warn("The model does not have any trainable weights.")

Batch 0: D\_loss=0.7068, D\_acc=0.0000, G\_loss=0.6797 Batch 100: D\_loss=1.0602, D\_acc=0.0000, G\_loss=0.2851 Batch 200: D\_loss=1.1610, D\_acc=0.0000, G\_loss=0.2271 Batch 300: D\_loss=1.2143, D\_acc=0.0000, G\_loss=0.2009 Batch 400: D\_loss=1.2459, D\_acc=0.0000, G\_loss=0.1861



Epoch 2/10

Batch 0: D\_loss=1.2613, D\_acc=0.0000, G\_loss=0.1792 Batch 100: D\_loss=1.2774, D\_acc=0.0000, G\_loss=0.1720 Batch 200: D\_loss=1.2891, D\_acc=0.0000, G\_loss=0.1668 Batch 300: D\_loss=1.2980, D\_acc=0.0000, G\_loss=0.1629 Batch 400: D\_loss=1.30

49,

D\_acc=0.0000, G\_loss=0.1598 Epoch 3/10

Batch 0: D\_loss=1.3091, D\_acc=0.0000, G\_loss=0.1580 Batch 100: D\_loss=1.3143, D\_acc=0.0000, G\_loss=0.1558