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

A. 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:

```
TensorFlow version: 2.16.1

--- Creating Tensors with Different Shapes ---

Scalar Tensor:
tf.Tensor(10, shape=(), dtype=int32)
Shape: tf.Tensor([], shape=(0,), dtype=int32)
Number of Dimensions (Rank): tf.Tensor(0, shape=(), dtype=int32)

Vector Tensor:
tf.Tensor([1 2 3 4 5], shape=(5,), dtype=int32)
Shape: tf.Tensor([5], shape=(1,), dtype=int32)
Number of Dimensions (Rank): tf.Tensor(1, shape=(), dtype=int32)

Matrix Tensor:
tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
Shape: tf.Tensor([3 2], shape=(2,), dtype=int32)
Number of Dimensions (Rank): tf.Tensor(2, shape=(), dtype=int32)
```

```
3-dimensional Tensor:
tf.Tensor(
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]], shape=(2, 2, 2), dtype=int32)
Shape: tf.Tensor([2 2 2], shape=(3,), dtype=int32)
Number of Dimensions (Rank): tf.Tensor(3, shape=(), dtype=int32)

4-dimensional Tensor (initialized with zeros):
tf.Tensor(
[[[[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]

  [[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]

  [[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]]


 [[[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]

  [[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]

  [[0. 0. 0. 0. 0.]
   [0. 0. 0. 0. 0.]]]], shape=(2, 3, 2, 5), dtype=float32)
Shape: tf.Tensor([2 3 2 5], shape=(4,), dtype=int32)
Number of Dimensions (Rank): tf.Tensor(4, shape=(), dtype=int32)
```

```
--- Creating Tensors with Different Data Types ---

Integer Tensor (tf.int32):
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
Data Type: <dtype: 'int32'>

Integer Tensor (tf.int64):
tf.Tensor([1 2 3], shape=(3,), dtype=int64)
Data Type: <dtype: 'int64'>

Floating-point Tensor (tf.float32):
tf.Tensor([1. 2. 3.], shape=(3,), dtype=float32)
Data Type: <dtype: 'float32'>

Floating-point Tensor (tf.float64):
tf.Tensor([1. 2. 3.], shape=(3,), dtype=float64)
Data Type: <dtype: 'float64'>

Boolean Tensor:
tf.Tensor([ True False  True], shape=(3,), dtype=bool)
Data Type: <dtype: 'bool'>

String Tensor:
tf.Tensor([b'hello' b'world'], shape=(2,), dtype=string)
Data Type: <dtype: 'string'>
```

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

Shaped Float Tensor (tf.float16):
tf.Tensor(
[[1.5 2.5]
 [3.5 4.5]], shape=(2, 2), dtype=float16)
Shape: tf.Tensor([2 2], shape=(2,), dtype=int32)
Data Type: <dtype: 'float16'>

Shaped Integer Tensor (initialized with zeros, tf.int32):
tf.Tensor(
[[0 0 0]
 [0 0 0]
 [0 0 0]], shape=(3, 3), dtype=int32)
Shape: tf.Tensor([3 3], shape=(2,), dtype=int32)
Data Type: <dtype: 'int32'>
```

B. 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:

```
--- Basic Tensor Operations ---

Tensor A:
tf.Tensor(
[[1. 2.]
 [3. 4.]], shape=(2, 2), dtype=float32)

Tensor B:
tf.Tensor(
[[5. 6.]
 [7. 8.]], shape=(2, 2), dtype=float32)

Scalar C:
tf.Tensor(2.0, shape=(), dtype=float32)

Addition (tensor_a + tensor_b):
tf.Tensor(
[[ 6.  8.]
 [10. 12.]], shape=(2, 2), dtype=float32)

Addition with scalar (tensor_a + scalar_c):
tf.Tensor(
[[3. 4.]
 [5. 6.]], shape=(2, 2), dtype=float32)

Subtraction (tensor_b - tensor_a):
tf.Tensor(
[[4. 4.]
 [4. 4.]], shape=(2, 2), dtype=float32)

Subtraction with scalar (tensor_a - scalar_c):
tf.Tensor(
[[-1.  0.]
 [ 1.  2.]], shape=(2, 2), dtype=float32)
```

```
Element-wise Multiplication (tensor_a * tensor_b):
tf.Tensor(
[[ 5. 12.]
 [21. 32.]], shape=(2, 2), dtype=float32)

Multiplication with scalar (tensor_a * scalar_c):
tf.Tensor(
[[2. 4.]
 [6. 8.]], shape=(2, 2), dtype=float32)

Element-wise Division (tensor_b / tensor_a):
tf.Tensor(
[[5.        3.       ]
 [2.3333333 2.       ]], shape=(2, 2), dtype=float32)

Division with scalar (tensor_a / scalar_c):
tf.Tensor(
[[0.5 1. ]
 [1.5 2. ]], shape=(2, 2), dtype=float32)

--- Important Considerations ---

Tensor D:
tf.Tensor([10. 20.], shape=(2,), dtype=float32)

Addition with broadcasting (tensor_a + tensor_d):
tf.Tensor(
[[11. 22.]
 [13. 24.]], shape=(2, 2), dtype=float32)
```

```
Tensor E:
tf.Tensor(
[[2.]
 [3.]], shape=(2, 1), dtype=float32)

Addition with broadcasting (tensor_a + tensor_e):
tf.Tensor(
[[3. 4.]
 [6. 7.]], shape=(2, 2), dtype=float32)

Addition with compatible data types (after casting):
tf.Tensor(
[[2. 4.]
 [6. 8.]], shape=(2, 2), dtype=float32)

--- Other Useful Operations ---

Matrix Multiplication (tf.matmul(tensor_a, tensor_b)):
tf.Tensor(
[[19. 22.]
 [43. 50.]], shape=(2, 2), dtype=float32)

Transpose of Tensor A (tf.transpose(tensor_a)):
tf.Tensor(
[[1. 3.]
 [2. 4.]], shape=(2, 2), dtype=float32)

Element-wise Power (tf.pow(tensor_a, 2)):
tf.Tensor(
[[ 1.  4.]
 [ 9. 16.]], shape=(2, 2), dtype=float32)

Element-wise Square Root (tf.sqrt(tensor_a)):
tf.Tensor(
[[1.        1.4142135]
 [1.7320508 2.       ]], shape=(2, 2), dtype=float32)
```

C. 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:

```
Original Tensor (shape: (3, 2, 3)):
[[[ 1  2  3]
  [ 4  5  6]]

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

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

--- Reshaping ---

Reshaped to 2x9 (shape: (2, 9)):
[[ 1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18]]

Reshaped to 1D (shape: (18,)):
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18]

Reshaped to 1x3x6 (shape: (1, 3, 6)):
[[[ 1  2  3  4  5  6]
  [ 7  8  9 10 11 12]
  [13 14 15 16 17 18]]]
```

```
--- Slicing ---

Slice: First 'outer' dimension (shape: (2, 3)):
[[1 2 3]
 [4 5 6]]

Slice: Second element of the inner dimension (shape: (3, 2)):
[[ 2  5]
 [ 8 11]
 [14 17]]

Slice: Element at [1, 0, 2]: 9

Slice with range (shape: (2, 1, 3)):
[[[1 2 3]]

 [[7 8 9]]]

Slice with step (shape: (3, 2, 2)):
[[[ 1  3]
  [ 4  6]]

 [[ 7  9]
  [10 12]]

 [[13 15]
  [16 18]]]
```

```
--- Indexing ---

Element at [2, 1, 0]: 16

Indexed elements using gather_nd (indices [[0, 0, 0], [1, 1, 2], [2, 0, 1]]):
[ 1 12 14]

Boolean mask (tensor > 10):
[[[False False False]
  [False False False]]

 [[False False False]
  [False  True  True]]

 [[ True  True  True]
  [ True  True  True]]]

Elements where mask is True (shape: (8,)):
[11 12 13 14 15 16 17 18]
```

D. **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:

```
Matrix Multiplication Demo
tf.Tensor(
[[1 2 3]
 [4 5 6]], shape=(2, 3), dtype=int32)
tf.Tensor(
[[ 7  8]
 [ 9 10]
 [11 12]], shape=(3, 2), dtype=int32)
Product: tf.Tensor(
[[ 58  64]
 [139 154]], shape=(2, 2), dtype=int32)
Matrix A:
[[8.854851  3.5752025]
 [6.110949  5.1205783]]


Eigen Vectors:
[[-0.5948931  0.8038049]
 [ 0.8038049  0.5948931]]

Eigen Values:
[ 0.59788704 13.3775425 ]
```

E. 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:

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 2)	6
dense_1 (Dense)	(None, 1)	3

Total params: 9 (36.00 B)

Trainable params: 9 (36.00 B)

```
Epoch 997/1000
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.7500 - loss: 0.5479
Epoch 998/1000
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.7500 - loss: 0.5477
Epoch 999/1000
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.7500 - loss: 0.5476
Epoch 1000/1000
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 31ms/step - accuracy: 0.7500 - loss: 0.5476
[array([[ 1.075752  , -0.60135233],
       [-1.0757387 , -0.3071198 ]], dtype=float32), array([0.0002241, 0.       ], dtype=float32), array([[1.5389476 ],
       [0.45380592]], dtype=float32), array([-0.37365782], dtype=float32)]
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step
[[0.40774074]
 [0.40765747]
 [0.7828263 ]
 [0.40774572]]
```

PRACTICAL 2

Linear Regression

A. 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:

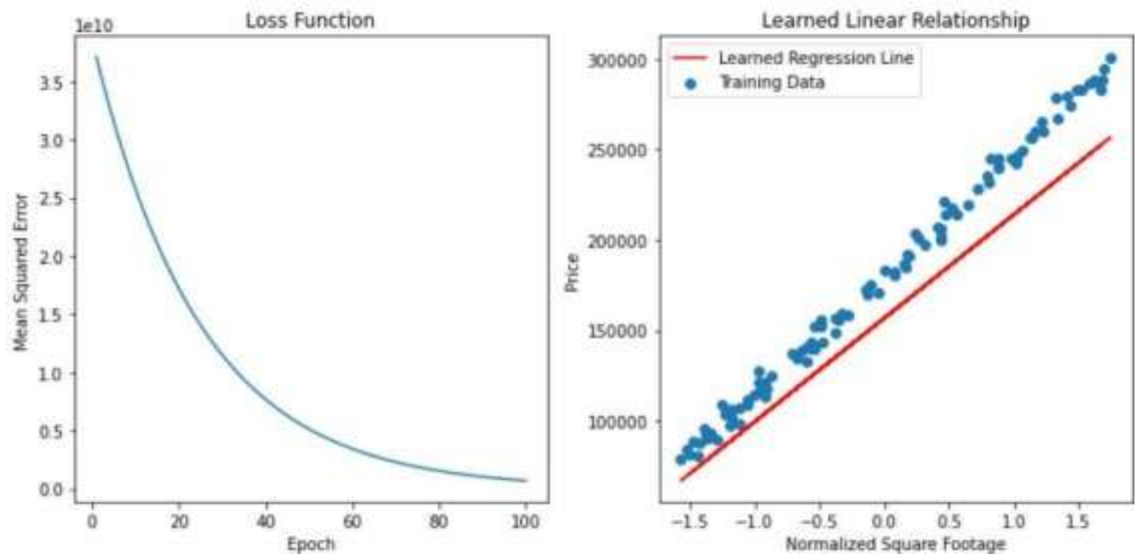
```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# 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:

```
Epoch 10, Loss: 25772652544.0000, Weight: 12058.41, Bias: 33079.69
Epoch 20, Loss: 17212729344.0000, Weight: 21911.01, Bias: 60108.20
Epoch 30, Loss: 11498058752.0000, Weight: 29961.29, Bias: 82192.46
Epoch 40, Loss: 7682897408.0000, Weight: 36538.96, Bias: 100236.92
Epoch 50, Loss: 5135865856.0000, Weight: 41913.39, Bias: 114980.55
Epoch 60, Loss: 3435447552.0000, Weight: 46304.69, Bias: 127027.17
Epoch 70, Loss: 2300233216.0000, Weight: 49892.71, Bias: 136870.16
Epoch 80, Loss: 1542356480.0000, Weight: 52824.38, Bias: 144912.56
Epoch 90, Loss: 1036394112.0000, Weight: 55219.76, Bias: 151483.78
Epoch 100, Loss: 698608832.0000, Weight: 57176.97, Bias: 156852.94
```




```
Predictions on new data points:
Square Footage: 750, Predicted Price: $98224.16
Square Footage: 1500, Predicted Price: $194807.45
Square Footage: 2200, Predicted Price: $284951.88
```

PRACTICAL 3

Convolutional Neural Networks (Classification)

A. Implementing deep neural network for performing binary classification task

CODE:

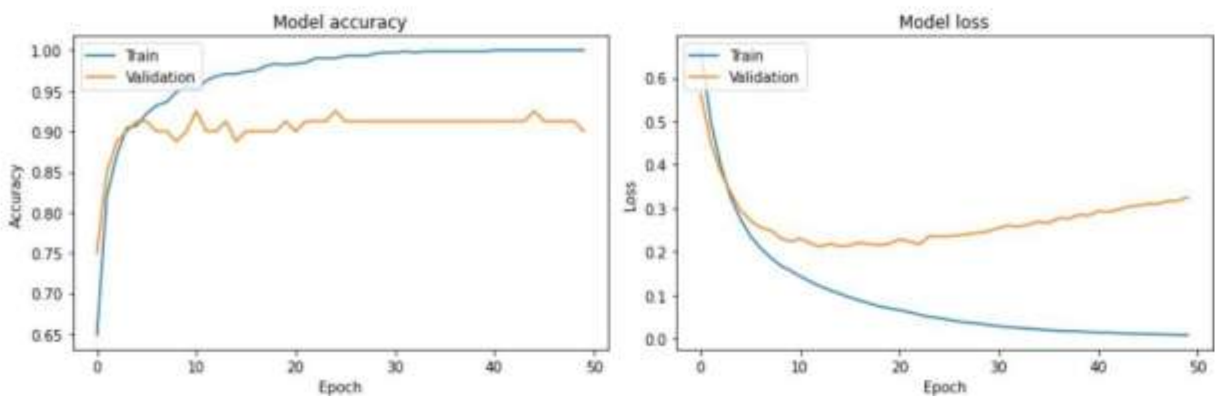
```
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification
import numpy as np
import matplotlib.pyplot as plt



# 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:

```
Test Loss: 0.2032
Test Accuracy: 0.9500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 124ms/step

Predictions on new data:
Sample: 1, Predicted Probability: 0.0000, Predicted Class: 0, True Class: 0
Sample: 2, Predicted Probability: 1.0000, Predicted Class: 1, True Class: 1
Sample: 3, Predicted Probability: 0.1400, Predicted Class: 0, True Class: 1
Sample: 4, Predicted Probability: 0.0018, Predicted Class: 0, True Class: 0
Sample: 5, Predicted Probability: 1.0000, Predicted Class: 1, True Class: 1
```



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

CODE:

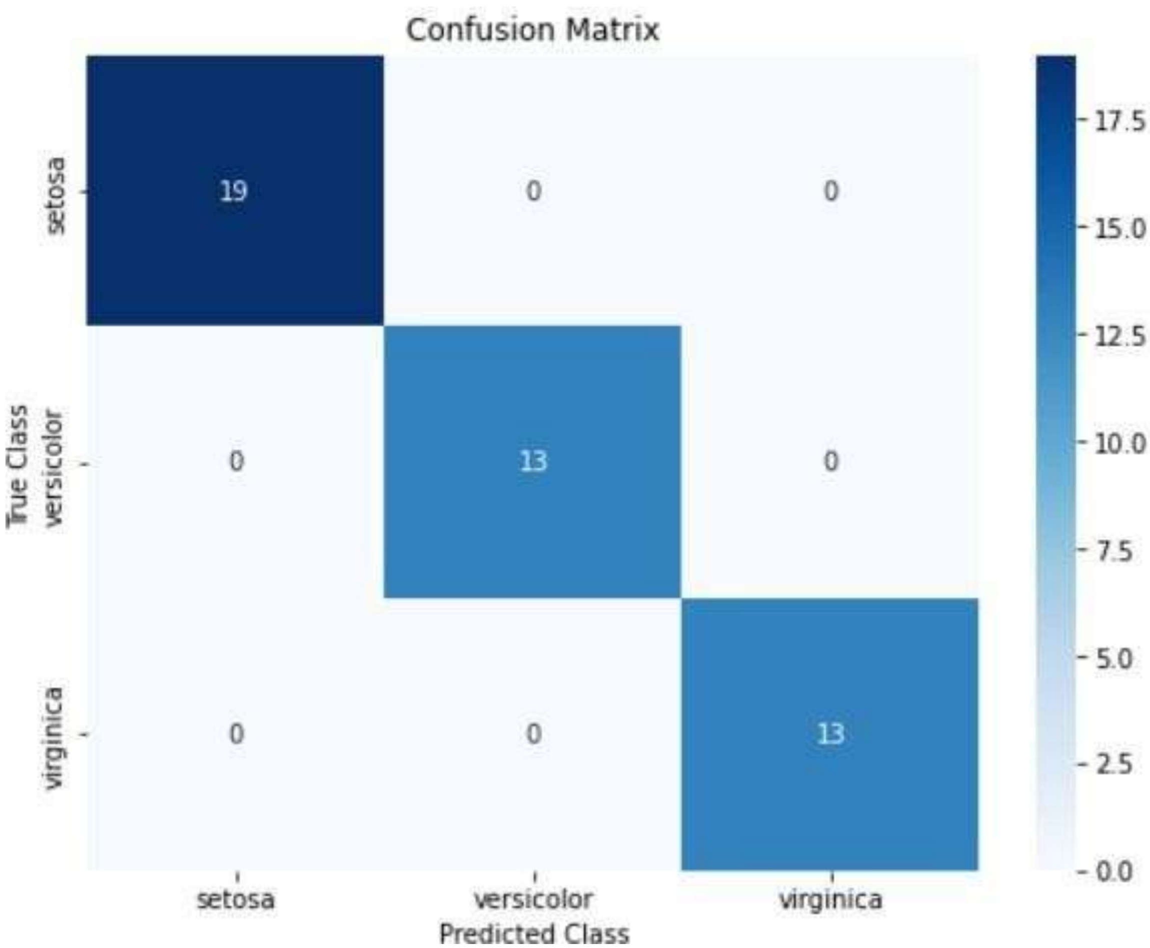
```
#pract3 b
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_iris
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np
import matplotlib.pyplot as plt
import 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
])
```

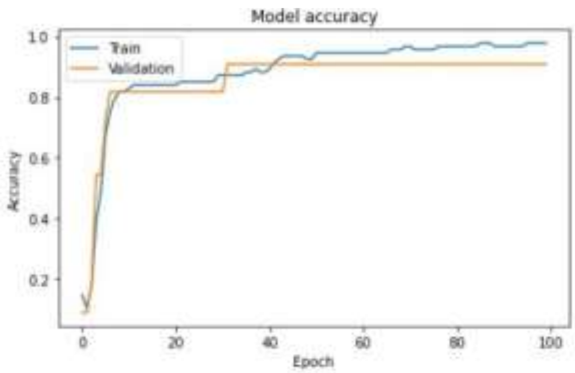
```
# 5. Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy', # Suitable loss for integer-encoded multiclass
              metrics=['accuracy'])

# 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))
```

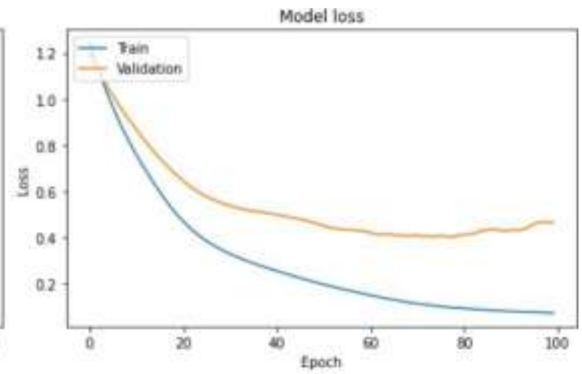
```
# Plot training & validation accuracy values
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')

# 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:

```
Test Loss: 0.0370
Test Accuracy: 1.0000
WARNING:tensorflow:5 out of the last 9 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_dis
tributed at 0x000002278797AB80> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings coul
d be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objec
ts instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing
=True option that can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/guide/function#controlli
ng_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/2 ━━━━━━━━━━━━━━━━━━━━ 0s 68ms/stepWARNING:tensorflow:6 out of the last 10 calls to <function TensorFlowTrainer.make_predict_
function.<locals>.one_step_on_data_distributed at 0x000002278797AB80> triggered tf.function retracing. Tracing is expensive and
the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with diffe
rent shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For
(2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to https://www.t
ensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more detail
s.
2/2 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        19
  versicolor       1.00      1.00      1.00        13
   virginica       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```








```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 156ms/step

Prediction for new data point [5.1, 3.5, 1.4, 0.2]:
Predicted Class Index: 0
Predicted Class Name: setosa
```

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:

Layer (type)	Output Shape	Param #	Connected to
input_layer_2 (InputLayer)	(None, 128, 128, 3)	0	-
conv2d (Conv2D)	(None, 128, 128, 32)	896	input_layer_2[0]…
conv2d_1 (Conv2D)	(None, 128, 128, 32)	9,248	conv2d[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 32)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 64, 64, 64)	18,496	max_pooling2d[0]…
conv2d_3 (Conv2D)	(None, 64, 64, 64)	36,928	conv2d_2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 64)	0	conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 32, 32, 128)	73,856	max_pooling2d_1[…
conv2d_5 (Conv2D)	(None, 32, 32, 128)	147,584	conv2d_4[0][0]

up_sampling2d (UpSampling2D)	(None, 64, 64, 128)	0	conv2d_5[0][0]
concatenate (Concatenate)	(None, 64, 64, 192)	0	up_sampling2d[0]… conv2d_3[0][0]
conv2d_6 (Conv2D)	(None, 64, 64, 64)	110,656	concatenate[0][0]
conv2d_7 (Conv2D)	(None, 64, 64, 64)	36,928	conv2d_6[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 128, 128, 64)	0	conv2d_7[0][0]
concatenate_1 (Concatenate)	(None, 128, 128, 96)	0	up_sampling2d_1[… conv2d_1[0][0]
conv2d_8 (Conv2D)	(None, 128, 128, 32)	27,680	concatenate_1[0]…
conv2d_9 (Conv2D)	(None, 128, 128, 32)	9,248	conv2d_8[0][0]
conv2d_10 (Conv2D)	(None, 128, 128, 1)	33	conv2d_9[0][0]

```
Total params: 471,553 (1.80 MB)
Trainable params: 471,553 (1.80 MB)
Non-trainable params: 0 (0.00 B)
Epoch 1/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 308s 12s/step - accuracy: 0.5000 - loss: 0.6934 - val_accuracy: 0.4997 - val_loss:
Epoch 2/10
```

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)

3. Defining the LSTM captioning model architecture...

Model: "functional_3"

Layer (type)	Output Shape	Param #	Connected to

Layer (type)	Output Shape	Param #	Connected to
input_layer_5 (InputLayer)	(None, 15)	0	-
input_layer_4 (InputLayer)	(None, 25088)	0	-
embedding_1 (Embedding)	(None, 15, 256)	3,072	input_layer_5[0]…
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 (NotEqual)	(None, 15)	0	input_layer_5[0]…
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)

4. 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"

Sample Image

--- 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:

Model: "functional_1"

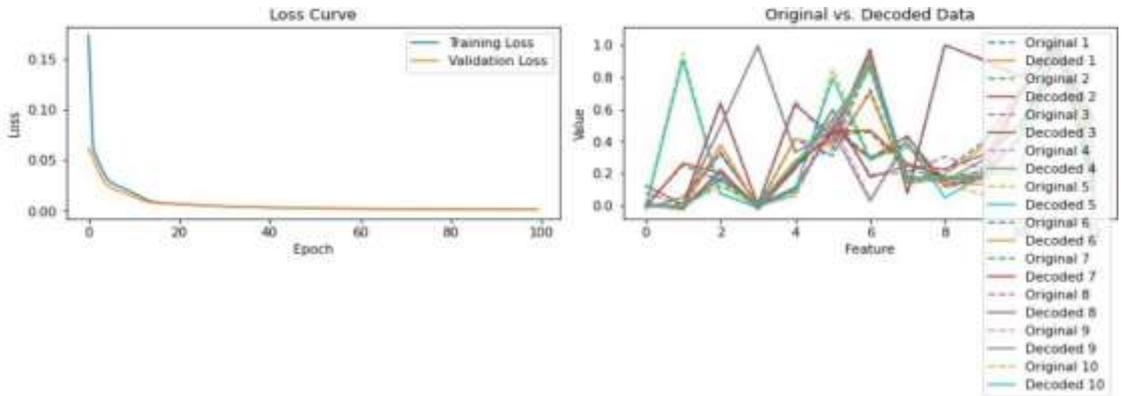
Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 13)	0
dense (Dense)	(None, 128)	1,792
dense_1 (Dense)	(None, 64)	8,256
dense_2 (Dense)	(None, 8)	520
dense_3 (Dense)	(None, 64)	576
dense_4 (Dense)	(None, 128)	8,320
dense_5 (Dense)	(None, 13)	1,677

Total params: 21,141 (82.58 KB)

Trainable params: 21,141 (82.58 KB)

Non-trainable params: 0 (0.00 B)

```
Encoded Data Shape: (102, 64)
Decoded Data Shape: (102, 13)
```




```
Example of using the encoder for feature extraction:
Original Data (first 3 samples):
[[9.29781490e-04 0.00000000e+00 1.22592593e-01 0.00000000e+00
  2.57201646e-01 5.19219036e-01 8.36251287e-01 1.37920687e-01
  1.73913043e-01 2.08015267e-01 4.25531915e-01 9.96469817e-01
  2.01710817e-01]
 [5.32556177e-04 4.00000000e-01 2.10000000e-01 1.00000000e+00
  1.27572016e-01 5.88773642e-01 3.08959835e-01 2.68075549e-01
  1.30434783e-01 1.27862595e-01 5.31914894e-01 1.00000000e+00
```

```
 0.7160788  0.         0.         0.25749835 0.         0.26289433
 0.         0.47291464 0.6195809  0.         0.         0.53989303
 0.48366085 0.         0.21680816 0.         0.         0.
 0.6354266  0.6458199  0.         0.         0.6609903  0.7616927
 0.         0.         0.26306838 0.4430914  0.6698895  0.6043509
 0.34611538 0.12486336 0.         0.         0.49383807 0.
 0.37970448 0.         0.08454479 0.25899053 0.59152085 0.4522654
 0.19903596 0.18127342 0.         0.         0.         0.
 0.         0.16238098 0.         0.         0.42601976 0.24224412
 0.43937856 0.42825297 0.         0.        ]
[0.10271323 0.         0.         0.         0.         0.
 0.73493326 0.         0.         0.9845544  0.         0.38261154
 0.         0.19893448 0.47234046 0.         0.5642811  0.26969925
 0.12076217 0.         0.28390563 0.         0.         0.
 0.816301   0.52392733 0.         0.         0.6025144  0.9349064
 0.         0.         0.15774986 0.31224614 1.0707345  1.0104039
 0.47338098 0.         0.         0.         0.06410319 0.
 0.5232336  0.         0.8620102  0.         1.0582277  0.1082851
 0.09026448 0.26699197 0.         0.         0.         0.
 0.01120614 0.         0.         0.         0.73214483 0.24195927
 0.22891009 0.6662339  0.         0.        ]
[0.23350123 0.         0.         0.         0.         0.
 1.1551294  0.         0.         0.6026196  0.         0.338678
 0.         0.46986842 0.86887    0.         0.08494157 0.28122598
 0.71264994 0.         0.279274   0.24394625 0.         0.
 1.1809355  1.1474549  0.         0.         0.7926923  1.0415461
 0.         0.         0.6322266  0.3892664  1.028541   0.9513966
 0.4440913  0.         0.         0.         0.6985725  0.
 0.7645968  0.         0.24454212 0.147512   0.74095684 0.6022995
 0.         0.00797881 0.         0.         0.         0.
 0.         0.4731131  0.         0.         0.62119997 0.13866049
 0.49856895 0.45271257 0.         0.        ]]
```

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
```

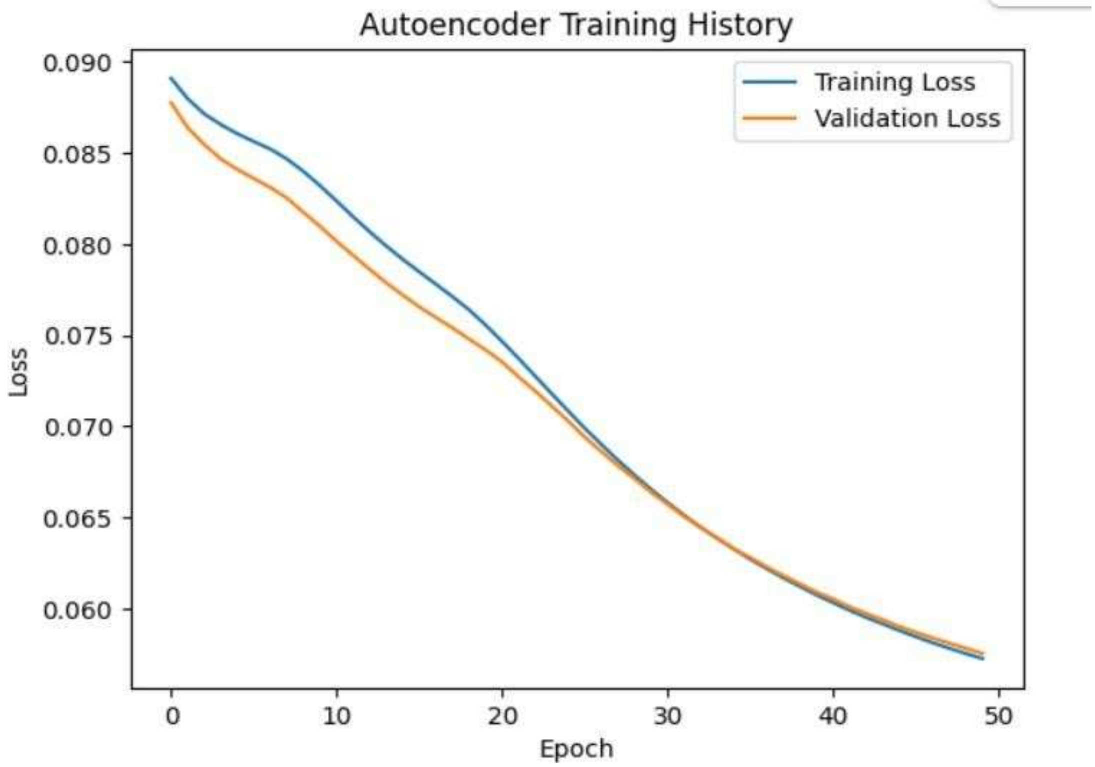
```
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Autoencoder Training History')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()

# 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()
```

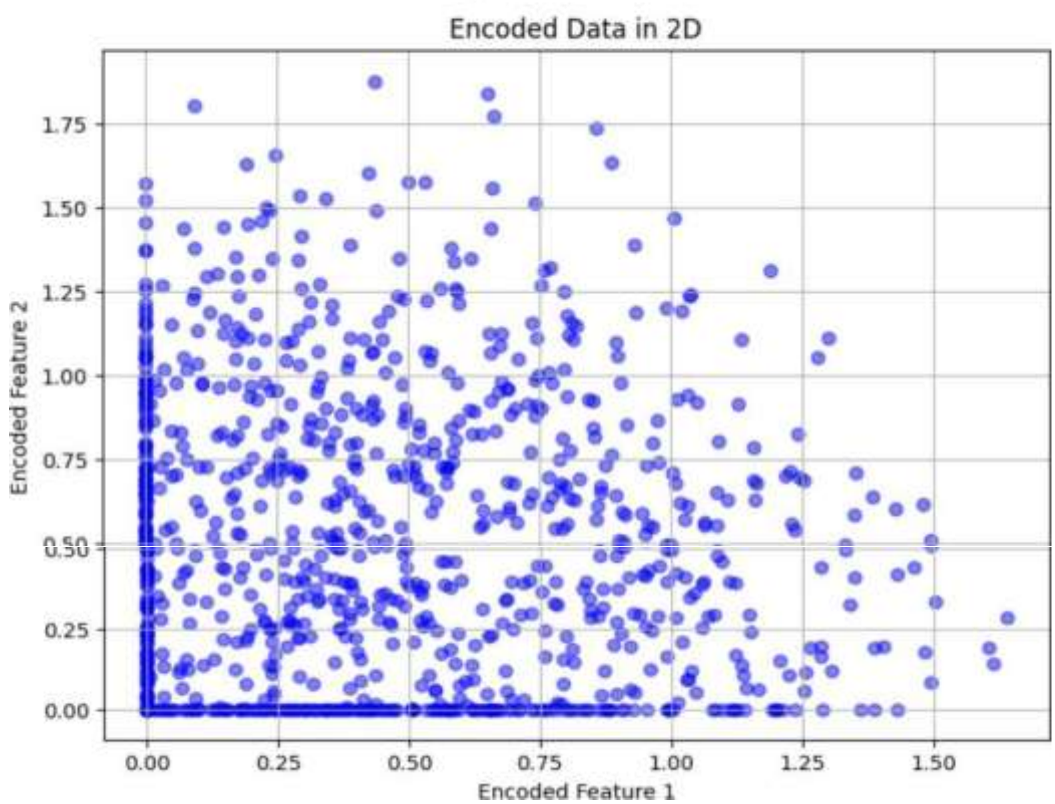
OUTPUT:

```
Epoch 1/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 2s 15ms/step - loss: 0.0887 - val_loss: 0.0878
Epoch 2/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0884 - val_loss: 0.0864
Epoch 3/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0880 - val_loss: 0.0855
Epoch 4/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0889 - val_loss: 0.0847
Epoch 5/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0854 - val_loss: 0.0841
Epoch 6/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0861 - val_loss: 0.0836
Epoch 7/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0847 - val_loss: 0.0831
Epoch 8/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0858 - val_loss: 0.0825
Epoch 9/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 0.0826 - val_loss: 0.0817
Epoch 10/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.0821 - val_loss: 0.0810
Epoch 11/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.0822 - val_loss: 0.0801
```

```
Epoch 49/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0574 - val_loss: 0.0578
Epoch 50/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0586 - val_loss: 0.0575
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step

Encoded Data (first 5 samples):
 [[1.2577895  0.11499357]
 [0.08732924 0.        ]
 [1.5056067  0.33298665]
 [0.4081761  0.15314406]
 [0.         0.5922005 ]]
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step

Decoded Data (first 5 samples):
 [[0.48465097 0.7306985  0.80285394 0.48153597 0.26811886]
 [0.48106903 0.3738676  0.59263885 0.65181035 0.5069611 ]
 [0.49658728 0.80358464 0.7854313  0.39029723 0.24739769]
 [0.48906764 0.48898676 0.6114002  0.5730562  0.4536658 ]
 [0.5154786  0.41365448 0.3298497  0.49626154 0.60983163]]
```

```
Original Normalized Data (first 5 samples):
 [[0.37584252 0.95098128 0.73280232 0.60048299 0.15597394]
 [0.15640395 0.05807107 0.86713485 0.60295089 0.70834659]
 [0.02044064 0.97018284 0.83336355 0.21237463 0.18179517]
 [0.18392593 0.30430678 0.52533265 0.4329975  0.2912625 ]
 [0.61412495 0.13950681 0.29246028 0.36711062 0.45619851]]

Test Loss (Mean Squared Error): 0.0575
```


Autoencoder Training History
Training Loss
Validation Loss
Loss
Epoch
0.090
0.085
0.080
0.075
0.070
0.065
0.060
0
10
20
30
40
50


Encoded Data in 2D
Encoded Feature 2
Encoded Feature 1
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00
1.25
1.50

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:

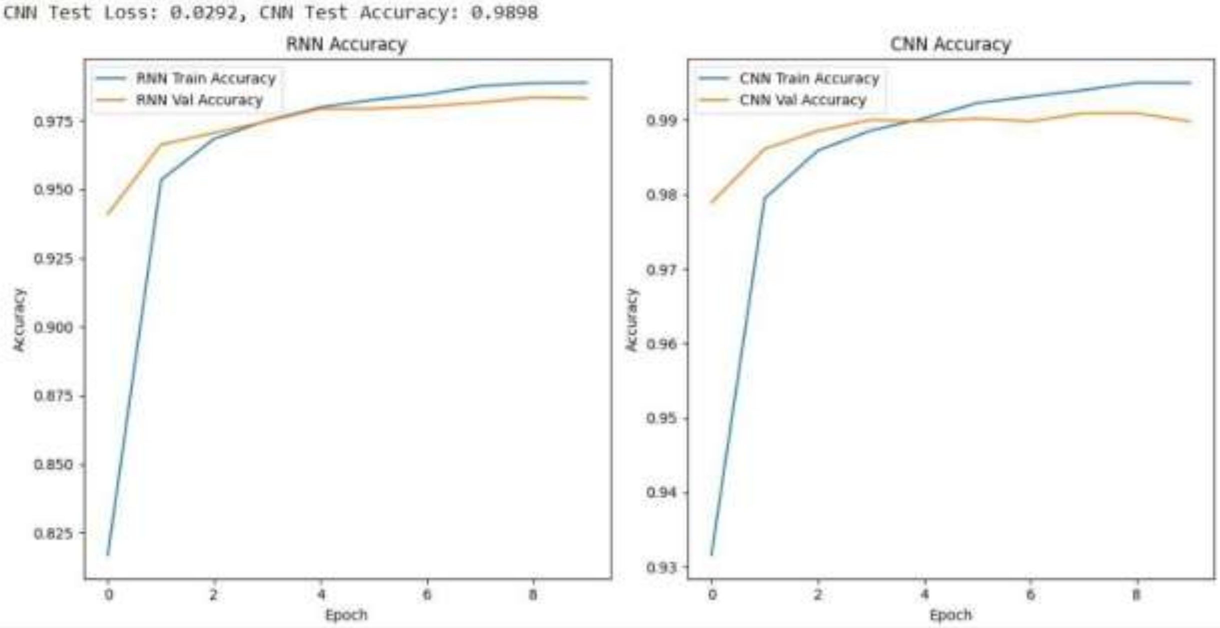
```
CNN Model Summary:
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
flatten (Flatten)	(None, 1600)	0
dropout_1 (Dropout)	(None, 1600)	0
dense_1 (Dense)	(None, 128)	204,928
dense_2 (Dense)	(None, 10)	1,290

Total params: 225,034 (879.04 KB)
Trainable params: 225,034 (879.04 KB)
Non-trainable params: 0 (0.00 B)

```
Training RNN Model:

Training CNN Model:

Evaluating RNN Model:
RNN Test Loss: 0.0550, RNN Test Accuracy: 0.9832

Evaluating CNN Model:
CNN Test Loss: 0.0292, CNN Test Accuracy: 0.9898
```



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")
```

```python
# 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)

2. Defining the Autoencoder model architecture...

3. 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)

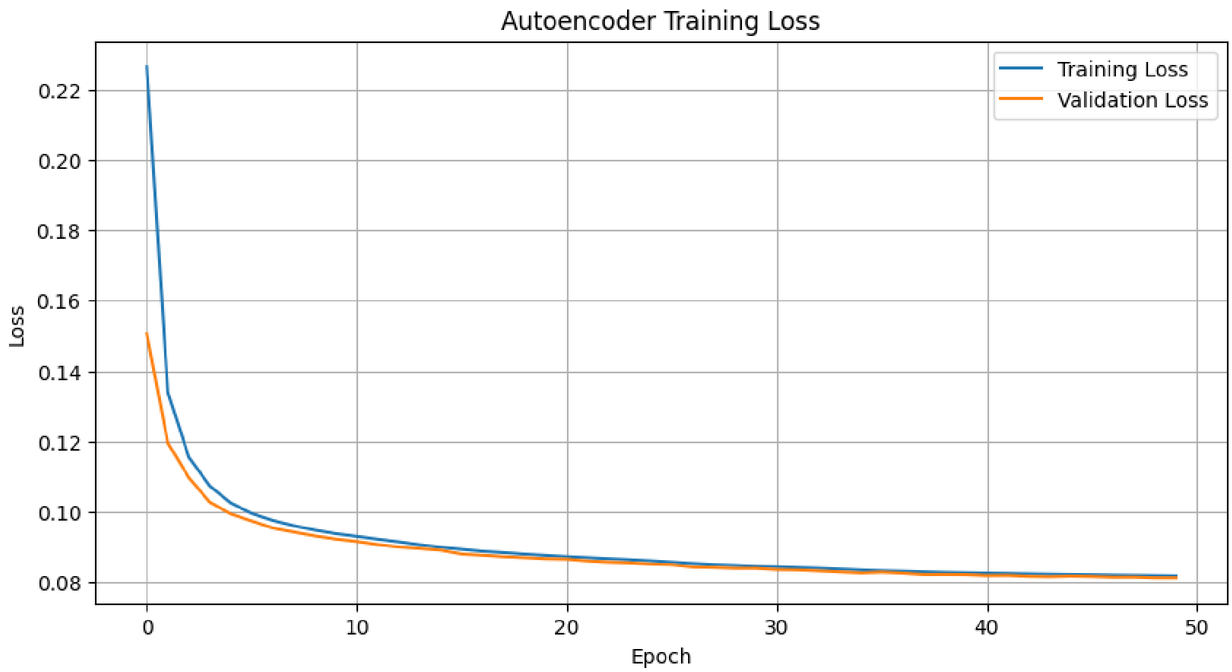
Trainable params: 209,968 (820.19 KB)

Non-trainable params: 0 (0.00 B)

4. 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 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - loss: 0.1089 - val_loss: 0.1027
Epoch 5/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 0.1034 - val_loss: 0.0994
Epoch 6/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 0.1002 - val_loss: 0.0974
Epoch 7/50
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 ━━━━━━━━━━━━━━━━━━━━ 5s 17ms/step - loss: 0.0822 - val_loss: 0.0817
Epoch 47/50
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
```

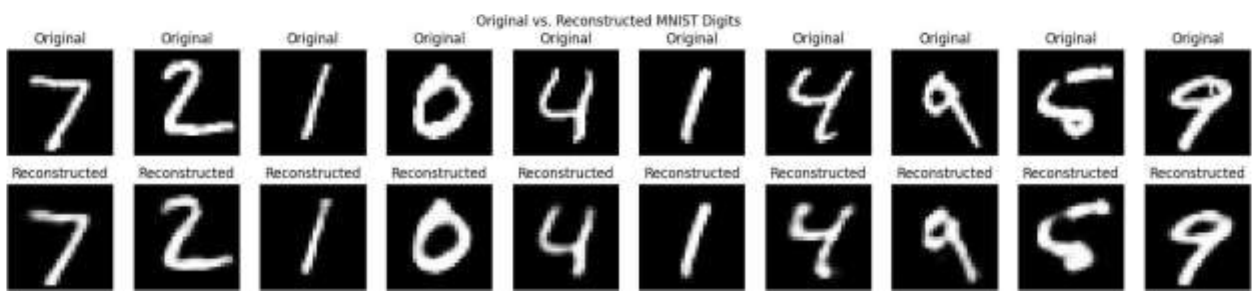
5. Plotting training loss history...



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,)

3. 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)

4. 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

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

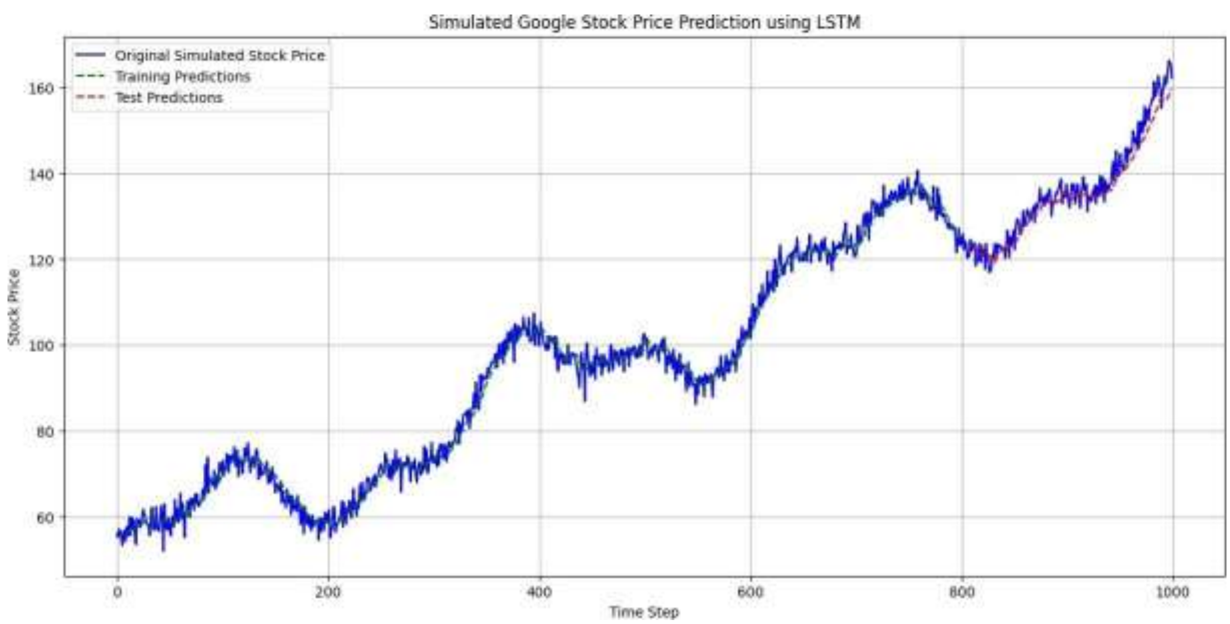
25/25 ━━━━━━━━━━━━━━━━━━━━ **0s** 10ms/step

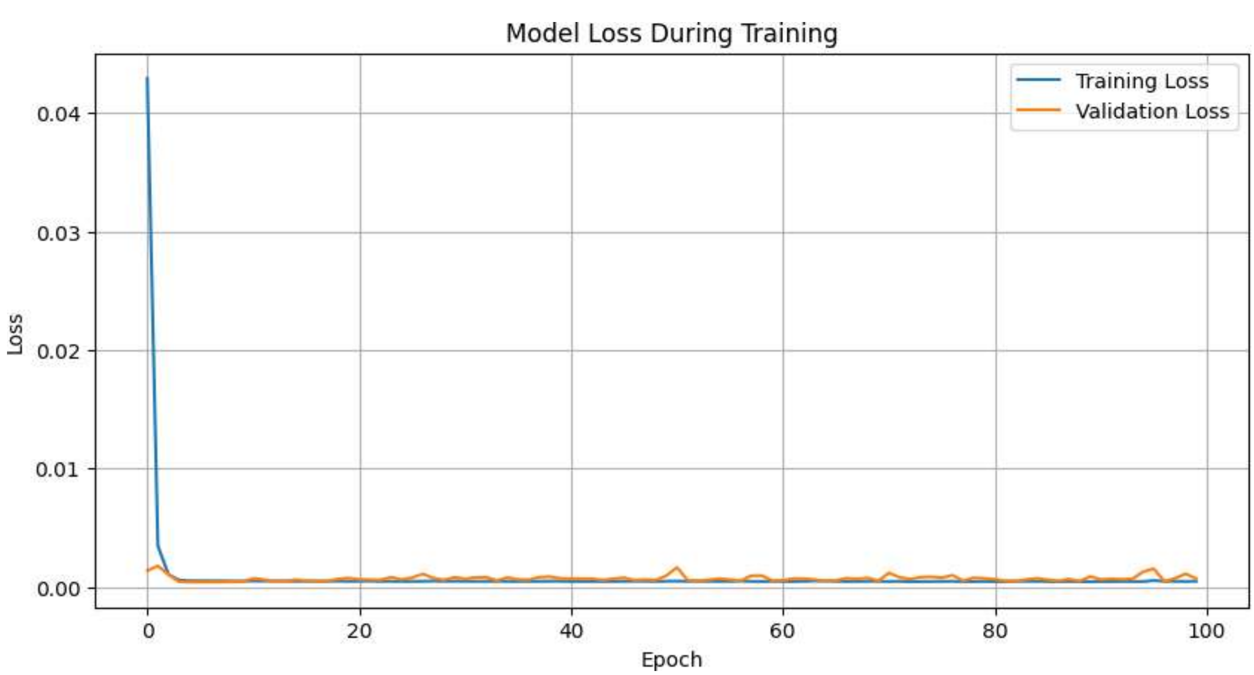
6/6 ━━━━━━━━━━━━━━━━━━━━ **0s** 6ms/step

Train RMSE: 2.40

Test RMSE: 3.04

6. 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)
```

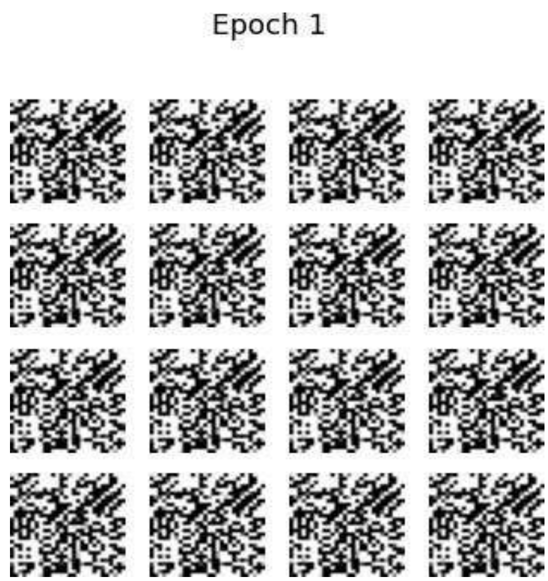
```
    # The discriminator tries to classify the generated image
    gan_output = discriminator(generated_image)

    # 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

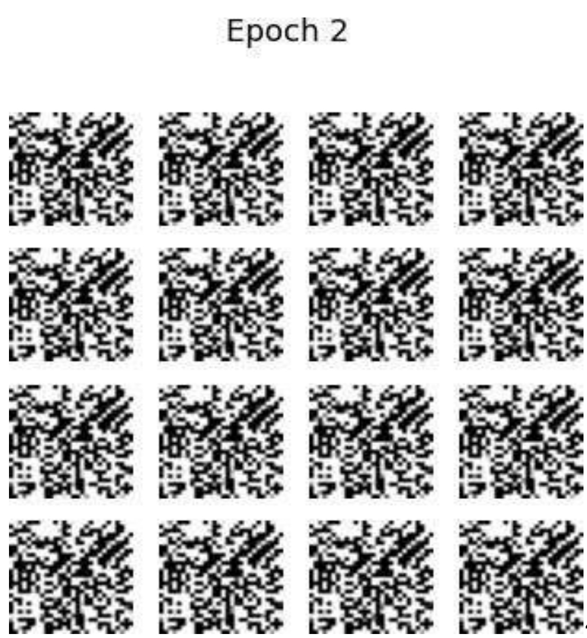
Batch 0: D_loss=1.2613, D_acc=0.0000, G_loss=0.1792

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