PRACTICAL JOURNAL

in

<u>Deep Learning</u>

<u>Submitted to</u>

Laxman Devram Sonawane College, Kalyan (W) 421301

in partial fulfilment for the award of the degree of

<u>Master of Science in Information Technology</u>

(Affiliated to Mumbai University)

Submitted by

<u>Karan Mahesh Katudiya</u>

Under the guidance of

Dr. Priyanka Pawar

Department of Information Technology
Kalyan, Maharashtra

Academic Year 2024-25

The Kalyan Wholesale Merchants Education Society's

Laxman Devram Sonawane College,

Kalyan (W) 421301

Department of Information Technology
Masters of Science – Part II

Certificate

This is to certify that **Mr. Karan Mahesh Katudiya**, Seat number______________, studying in Masters of Science in Information Technology Part II, Semester IV has satisfactorily completed the practical of "**Deep Learning** " as prescribed by University of Mumbai, during the academic year 2024-25.

Subject In-charge	Coordinator In-charge	ExternalExaminer

College Seal

INDEX

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

Practical 1 : Introduction to TensorFlow

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

Code :

```
import tensorflow as tf

# Create a scalar (0-D tensor)
scalar = tf.constant(42)
print("Scalar (0-D Tensor):", scalar)

# Create a vector (1-D tensor)
vector = tf.constant([1.5, 2.5, 3.5], dtype=tf.float32)
print("Vector (1-D Tensor):", vector)

# Create a matrix (2-D tensor)
matrix = tf.constant([[1, 2], [3, 4], [5, 6]], dtype=tf.int32)
print("Matrix (2-D Tensor):", matrix)

# Create a 3-D tensor
tensor_3d = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print("3-D Tensor:", tensor_3d)

# Create a tensor of all zeros
zeros_tensor = tf.zeros([2, 3])
print("Zeros Tensor:", zeros_tensor)

# Create a tensor of all ones
ones_tensor = tf.ones([3, 2, 2])
print("Ones Tensor:", ones_tensor)
```

 # Create a tensor with random values

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

print("Random Tensor:", random_tensor)

 # Get the shape and data type of a tensor

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

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

Output :

```
Scalar (0-D Tensor): tf.Tensor(42, shape=(), dtype=int32)
Vector (1-D Tensor): tf.Tensor([1.5 2.5 3.5], shape=(3,), dtype=float32)
Matrix (2-D Tensor): tf.Tensor(
[[1 2]
 [3 4]
 [5 6]], shape=(3, 2), dtype=int32)
3-D Tensor: tf.Tensor(
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]], shape=(2, 2, 2), dtype=int32)
Zeros Tensor: tf.Tensor(
[[0. 0. 0.]
 [0. 0. 0.]], shape=(2, 3), dtype=float32)
Ones Tensor: tf.Tensor(
[[[1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]]], shape=(3, 2, 2), dtype=float32)
Random Tensor: tf.Tensor(
[[-1.2160594  0.9616411]
 [-1.099461   0.8020662]], shape=(2, 2), dtype=float32)
Shape of matrix: (3, 2)
Data type of vector: <dtype: 'float32'>
```

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

Code :

```
import tensorflow as tf

# Define two tensors
a = tf.constant([3, 6, 9], dtype=tf.int32)
b = tf.constant([2, 4, 6], dtype=tf.int32)

# Perform basic arithmetic operations
addition = tf.add(a, b)
subtraction = tf.subtract(a, b)
multiplication = tf.multiply(a, b)
division = tf.divide(a, b)

# Display the results
print("Tensor A:", a.numpy())
print("Tensor B:", b.numpy())
print("Addition:", addition.numpy())
print("Subtraction:", subtraction.numpy())
print("Multiplication:", multiplication.numpy())
print("Division:", division.numpy())

# More tensor operations
squared = tf.square(a)
sqrt_b = tf.sqrt(tf.cast(b, tf.float32))
dot_product = tf.tensordot(a, b, axes=1)
```

```
print("Squared A:", squared.numpy())
print("Square root of B:", sqrt_b.numpy())
print("Dot product of A and B:", dot_product.numpy())
```

Output :

```
Tensor A: [3 6 9]
Tensor B: [2 4 6]
Addition: [ 5 10 15]
Subtraction: [1 2 3]
Multiplication: [ 6 24 54]
Division: [1.5 1.5 1.5]
Squared A: [ 9 36 81]
Square root of B: [1.4142135 2.        2.4494898]
Dot product of A and B: 84
```

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

Code :

```python
import tensorflow as tf

# Create a sample tensor
original_tensor = tf.constant([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Original Tensor:\n", original_tensor.numpy())

# Reshape the tensor
reshaped_tensor = tf.reshape(original_tensor, (1, 9))
print("\nReshaped Tensor (1x9):\n", reshaped_tensor.numpy())

# Slice the tensor (Extract rows 1 and 2, columns 1 and 2)
sliced_tensor = original_tensor[1:3, 1:3]
print("\nSliced Tensor (Rows 1-2, Columns 1-2):\n", sliced_tensor.numpy())

# Indexing (Extract specific elements)
first_element = original_tensor[0, 0]
last_row = original_tensor[-1]
print("\nFirst Element (0,0):", first_element.numpy())
print("Last Row:", last_row.numpy())

# Use tf.gather to extract specific indices
gathered_elements = tf.gather(original_tensor, [0, 2], axis=0)
print("\nGathered Rows 0 and 2:\n", gathered_elements.numpy())

# Use tf.boolean_mask to extract elements with a condition
```

```
masked_elements = tf.boolean_mask(original_tensor, original_tensor > 4)
print("\nElements Greater than 4:\n", masked_elements.numpy())
```

Output :

```
Original Tensor:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Reshaped Tensor (1x9):
 [[1 2 3 4 5 6 7 8 9]]

Sliced Tensor (Rows 1-2, Columns 1-2):
 [[5 6]
 [8 9]]

First Element (0,0): 1
Last Row: [7 8 9]

Gathered Rows 0 and 2:
 [[1 2 3]
 [7 8 9]]

Elements Greater than 4:
 [5 6 7 8 9]
```

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

Code :

```
import tensorflow as tf

import numpy as np


# Define two matrices

matrix_a = tf.constant([[3, 4], [5, 6]], dtype=tf.float32)

matrix_b = tf.constant([[7, 8], [9, 10]], dtype=tf.float32)


# Perform matrix multiplication

matrix_product = tf.matmul(matrix_a, matrix_b)


# Display the result of matrix multiplication

print("Matrix A:", matrix_a.numpy())

print("Matrix B:", matrix_b.numpy())

print("Matrix Product (A * B):", matrix_product.numpy())


# Finding eigenvalues and eigenvectors

matrix_c = tf.constant([[4, -2], [1, 1]], dtype=tf.float32)


# Convert TensorFlow tensor to NumPy array for eigenvalue computation

matrix_c_np = matrix_c.numpy()

eigenvalues, eigenvectors = np.linalg.eig(matrix_c_np)


# Display eigenvalues and eigenvectors

print("Matrix C:", matrix_c_np)

print("Eigenvalues:", eigenvalues)
```

```
print("Eigenvectors:\n", eigenvectors)
```

Output :

```
Matrix A: [[3. 4.]
 [5. 6.]]
Matrix B: [[ 7.  8.]
 [ 9. 10.]]
Matrix Product (A * B): [[ 57.  64.]
 [ 89. 100.]]
Matrix C: [[ 4. -2.]
 [ 1.  1.]]
Eigenvalues: [3. 2.]
Eigenvectors:
 [[0.8944272  0.70710677]
 [0.4472136  0.70710677]]
```

1b. Program to solve the XOR problem

Code :

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np

# Define XOR inputs and outputs
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]], dtype=np.float32)
y = np.array([[0], [1], [1], [0]], dtype=np.float32)

# Define a simple neural network model
model = keras.Sequential([
    layers.Dense(4, activation='relu', input_shape=(2,)),
    layers.Dense(4, activation='relu'),
    layers.Dense(1, activation='sigmoid')])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

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

# Evaluate the model
loss, accuracy = model.evaluate(X, y)
print(f"Final Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

# Make predictions
predictions = model.predict(X)
```

```
print("\nPredictions:")
for i, p in enumerate(predictions):
    print(f"Input: {X[i]} => Predicted Output: {p[0]:.4f} => Rounded: {int(np.round(p[0]))}")
```

Output :

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 191ms/step - accuracy: 0.7500 - loss: 0.5135
Final Loss: 0.5135, Accuracy: 0.7500
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 74ms/step

Predictions:
Input: [0. 0.] => Predicted Output: 0.5979 => Rounded: 1
Input: [0. 1.] => Predicted Output: 0.5975 => Rounded: 1
Input: [1. 0.] => Predicted Output: 0.5979 => Rounded: 1
Input: [1. 1.] => Predicted Output: 0.1074 => Rounded: 0
```

Practical 2 : Linear Regression

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

Code :

```
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt


# Generate synthetic data

np.random.seed(42)

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

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


# Define a simple linear regression model

model = tf.keras.Sequential([

   tf.keras.layers.Dense(1, input_shape=(1,))])


# Compile the model

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


# Train the model

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


# Get the model's weights

W, b = model.layers[0].get_weights()

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


# Make predictions
```

y_pred = model.predict(X)

 # Plot the results

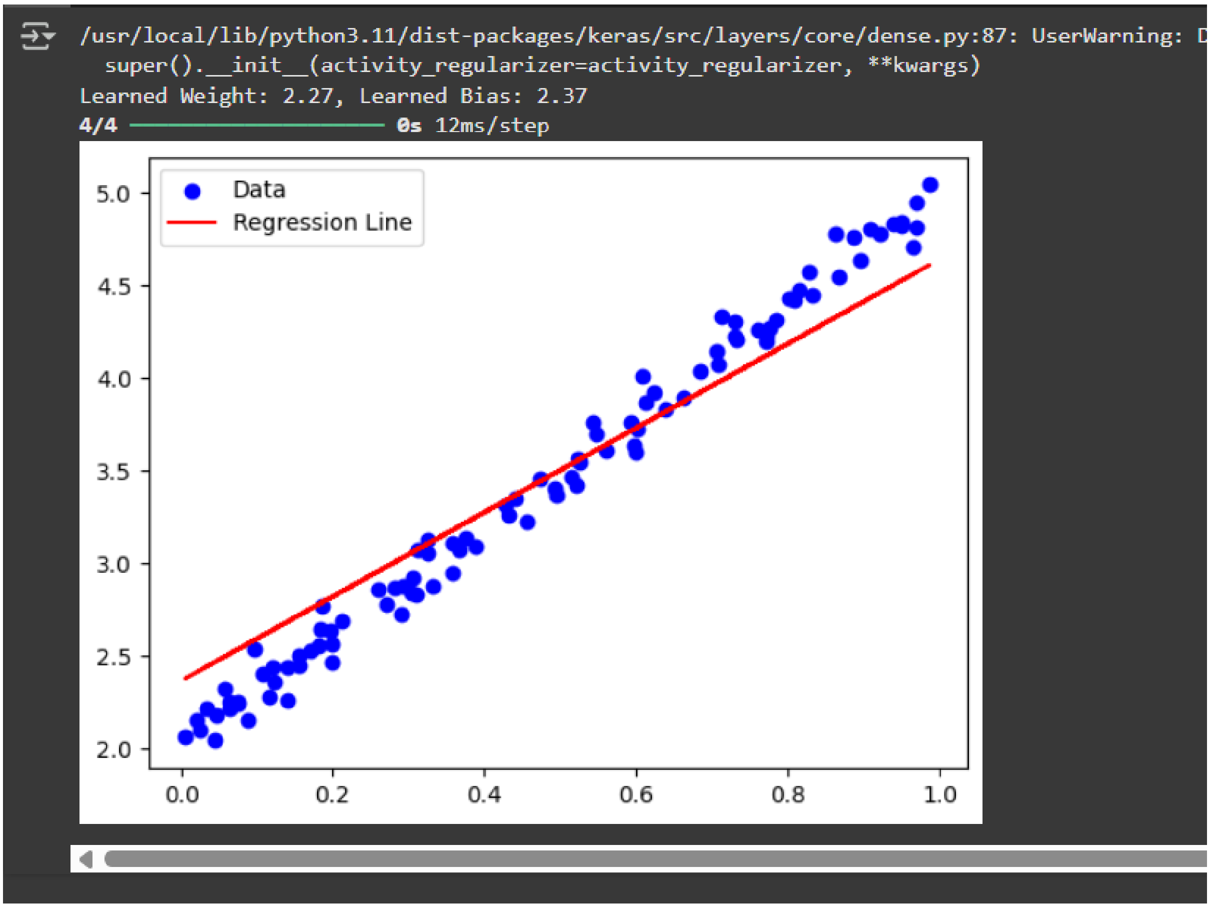
plt.scatter(X, y, color='blue', label='Data')

plt.plot(X, y_pred, color='red', label='Regression Line')

plt.legend()

plt.show()

Output :



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

Code :

```
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt


# Toy dataset: square footage vs. housing prices

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

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


# Reshape data

X = square_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)


# Define a simple linear regression model

model = tf.keras.Sequential([

   tf.keras.layers.Dense(1, input_shape=(1,))])


# Compile the model

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


# Train the model

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


# Get the model's weights

W, b = model.layers[0].get_weights()
```

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

 # Make predictions

y_pred = model.predict(X)

 # Plot the results

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

plt.plot(X, y_pred, color='red', label='Regression Line')

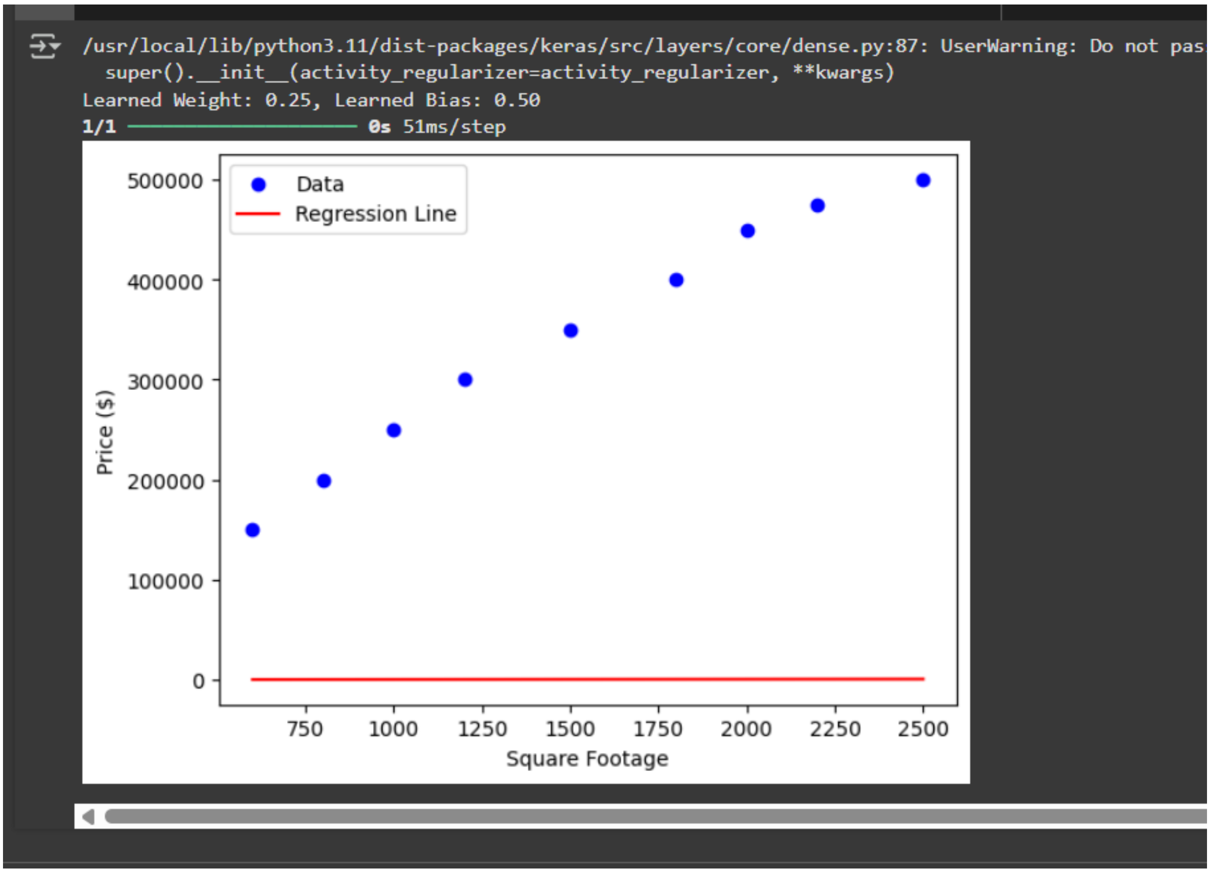
plt.xlabel('Square Footage')

plt.ylabel('Price ($)')

plt.legend()

plt.show()

Output :



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

Code :

```
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt


# Toy dataset: square footage vs. housing prices

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

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


# Reshape data

X = square_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)


# Define a simple linear regression model

model = tf.keras.Sequential([

   tf.keras.layers.Dense(1, input_shape=(1,))])


# Compile the model

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

# Train the model

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


# Get the model's weights

W, b = model.layers[0].get_weights()

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


# Make predictions
```

```
y_pred = model.predict(X)
```

 # Plot the results

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

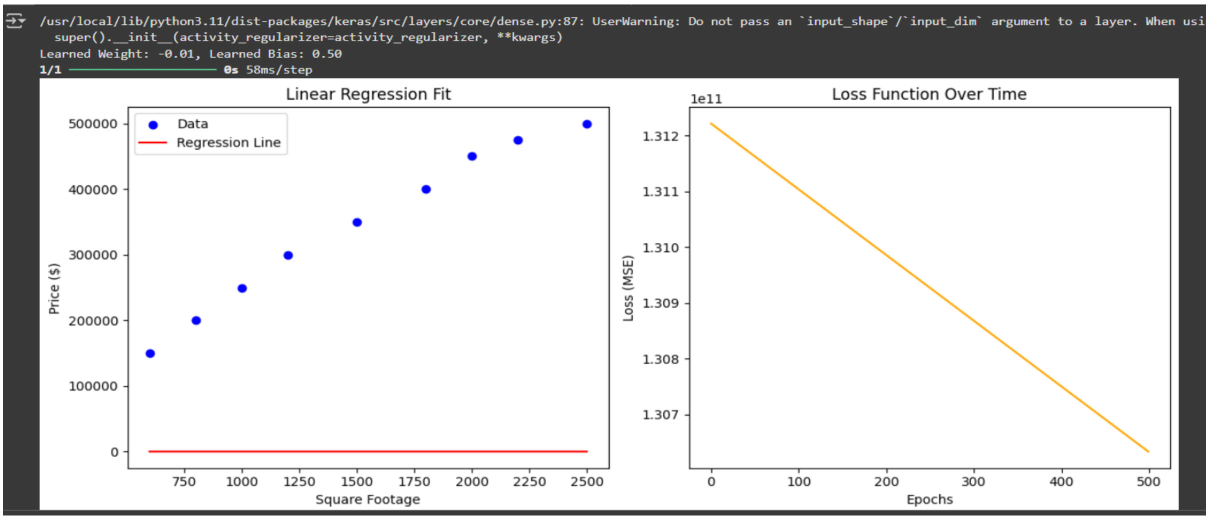
 # Plot the data and regression line

```
plt.subplot(1, 2, 1)
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, y_pred, color='red', label='Regression Line')
plt.xlabel('Square Footage')
plt.ylabel('Price ($)')
plt.legend()
plt.title('Linear Regression Fit')
```

 # Plot the loss function

```
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], color='orange')
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.title('Loss Function Over Time')
plt.tight_layout()
plt.show()
```

Output :



2-a4. Make predictions on new data points

Code :

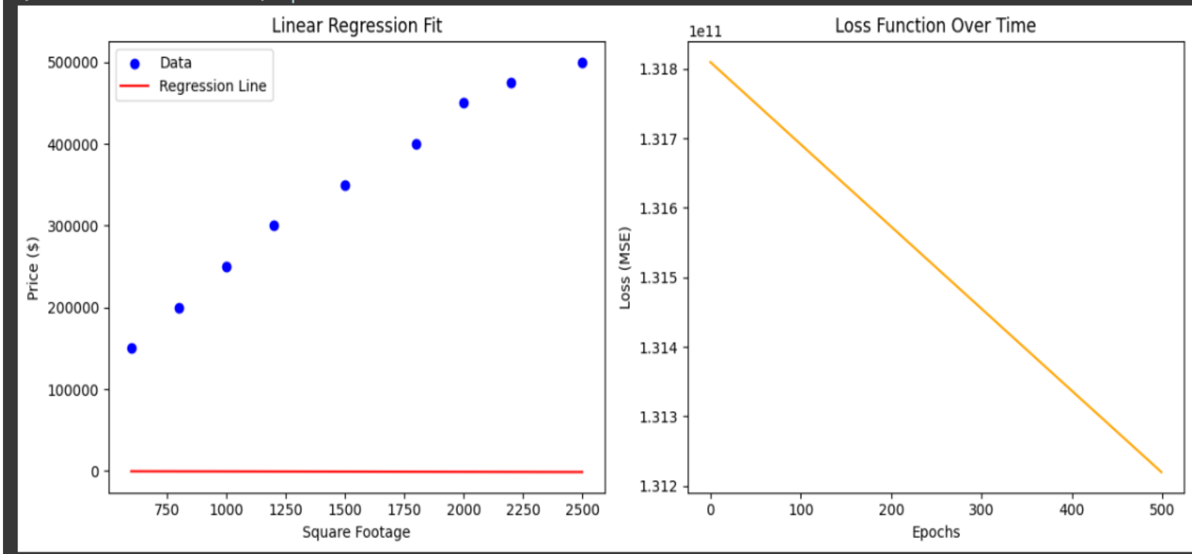
```
import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt


# Toy dataset: square footage vs. housing prices

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

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


# Reshape data

X = square_feet.reshape(-1, 1)

y = prices.reshape(-1, 1)


# Define a simple linear regression model

model = tf.keras.Sequential([

    tf.keras.layers.Dense(1, input_shape=(1,))])


# Compile the model

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


# Train the model

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


# Get the model's weights

W, b = model.layers[0].get_weights()

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

```
# Make predictions
y_pred = model.predict(X)

# Visualize the results
plt.figure(figsize=(12, 5))

# Plot the data and regression line
plt.subplot(1, 2, 1)
plt.scatter(X, y, color='blue', label='Data')
plt.plot(X, y_pred, color='red', label='Regression Line')
plt.xlabel('Square Footage')
plt.ylabel('Price ($)')
plt.legend()
plt.title('Linear Regression Fit')

# Plot the loss function
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], color='orange')
plt.xlabel('Epochs')
plt.ylabel('Loss (MSE)')
plt.title('Loss Function Over Time')
plt.tight_layout()
plt.show()

# Make predictions on new data points
new_data = np.array([[1600], [2100], [3000]], dtype=np.float32)
new_predictions = model.predict(new_data)

print("\nPredictions on New Data:")
for sqft, price in zip(new_data.flatten(), new_predictions.flatten()):
```

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

Output :

```
/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 Sequ
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Learned Weight: -0.50, Learned Bias: 0.50
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 82ms/step
```




```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 76ms/step
```

```
Predictions on New Data:
Square Footage: 1600.0, Predicted Price: $-805.47
Square Footage: 2100.0, Predicted Price: $-1057.33
Square Footage: 3000.0, Predicted Price: $-1510.69
```

Practical 3 : Convolutional Neural Networks (Classification)

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

Code :

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import cifar10
import matplotlib.pyplot as plt

# Load and preprocess data
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# For binary classification, let's classify 'airplane' (class 0) vs 'automobile' (class 1)
# Reshape y_trrain & y_test to 1D arrays
y_train = y_train.reshape(-1)
y_test = y_test.reshape(-1)

X_train = X_train[(y_train == 0) | (y_train == 1)].astype('float32') / 255.0
X_test = X_test[(y_test == 0) | (y_test == 1)].astype('float32') / 255.0

y_train = y_train[(y_train == 0) | (y_train == 1)]
y_test = y_test[(y_test == 0) | (y_test == 1)]

# Build CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
```

layers.MaxPooling2D((2, 2)),

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

layers.Flatten(),

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

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

 # Compile the model

model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

 # Train the model

history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test), verbose=2)

 # Evaluate the model

loss, accuracy = model.evaluate(X_test, y_test)

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

 # Plot training history

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

plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

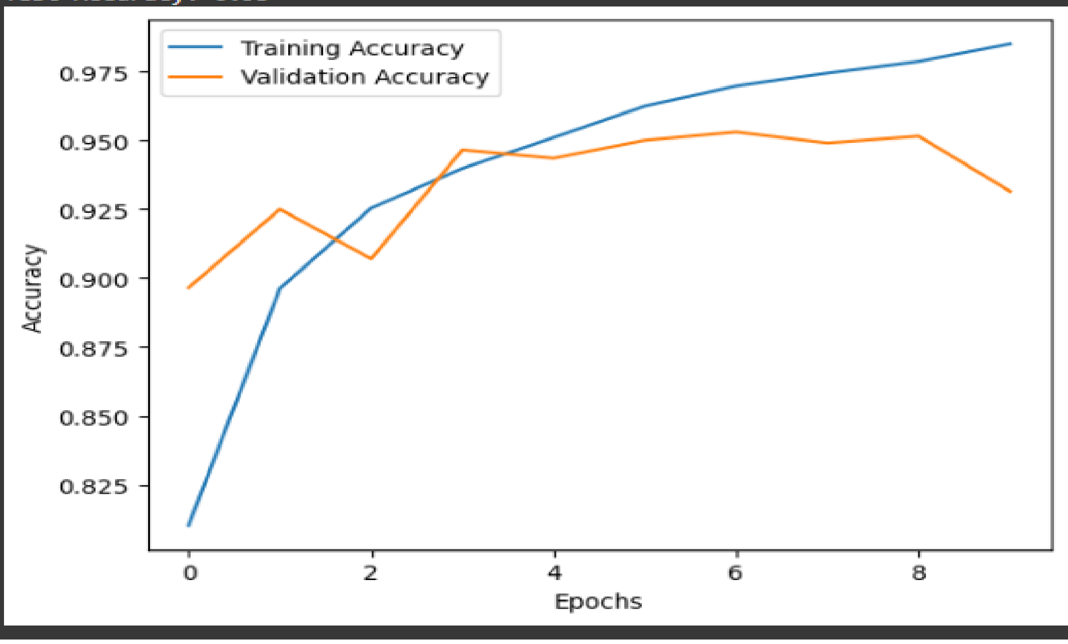
plt.legend()

plt.show()

Output :

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv.py:107: UserWarning: Do not pass an
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
313/313 - 16s - 52ms/step - accuracy: 0.8103 - loss: 0.4080 - val_accuracy: 0.8965 - val_loss: 0.2718
Epoch 2/10
313/313 - 20s - 63ms/step - accuracy: 0.8962 - loss: 0.2507 - val_accuracy: 0.9250 - val_loss: 0.1876
Epoch 3/10
313/313 - 14s - 44ms/step - accuracy: 0.9254 - loss: 0.1894 - val_accuracy: 0.9070 - val_loss: 0.2266
Epoch 4/10
313/313 - 21s - 66ms/step - accuracy: 0.9398 - loss: 0.1529 - val_accuracy: 0.9465 - val_loss: 0.1405
Epoch 5/10
313/313 - 21s - 66ms/step - accuracy: 0.9510 - loss: 0.1233 - val_accuracy: 0.9435 - val_loss: 0.1526
Epoch 6/10
313/313 - 20s - 64ms/step - accuracy: 0.9624 - loss: 0.0994 - val_accuracy: 0.9500 - val_loss: 0.1584
Epoch 7/10
313/313 - 21s - 66ms/step - accuracy: 0.9697 - loss: 0.0846 - val_accuracy: 0.9530 - val_loss: 0.1345
Epoch 8/10
313/313 - 21s - 66ms/step - accuracy: 0.9744 - loss: 0.0689 - val_accuracy: 0.9490 - val_loss: 0.1546
Epoch 9/10
313/313 - 21s - 68ms/step - accuracy: 0.9786 - loss: 0.0595 - val_accuracy: 0.9515 - val_loss: 0.1363
Epoch 10/10
313/313 - 20s - 63ms/step - accuracy: 0.9850 - loss: 0.0421 - val_accuracy: 0.9315 - val_loss: 0.2261
63/63 ━━━━━━━━━━━━━━━━━━━━ 1s 13ms/step - accuracy: 0.9363 - loss: 0.1993
Test Accuracy: 0.93
```

Test Accuracy: 0.93



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

Code :

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load and preprocess data
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the data
X_train = X_train.reshape(-1, 28 * 28).astype('float32') / 255.0
X_test = X_test.reshape(-1, 28 * 28).astype('float32') / 255.0

# Define a deep feed-forward network model
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(28 * 28,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')])
# Compile the model
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test,
y_test), verbose=2)

# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy:.2f}")
```

 # Make predictions on new data

predictions = model.predict(X_test[:5])

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

 # Plot training history

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

plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

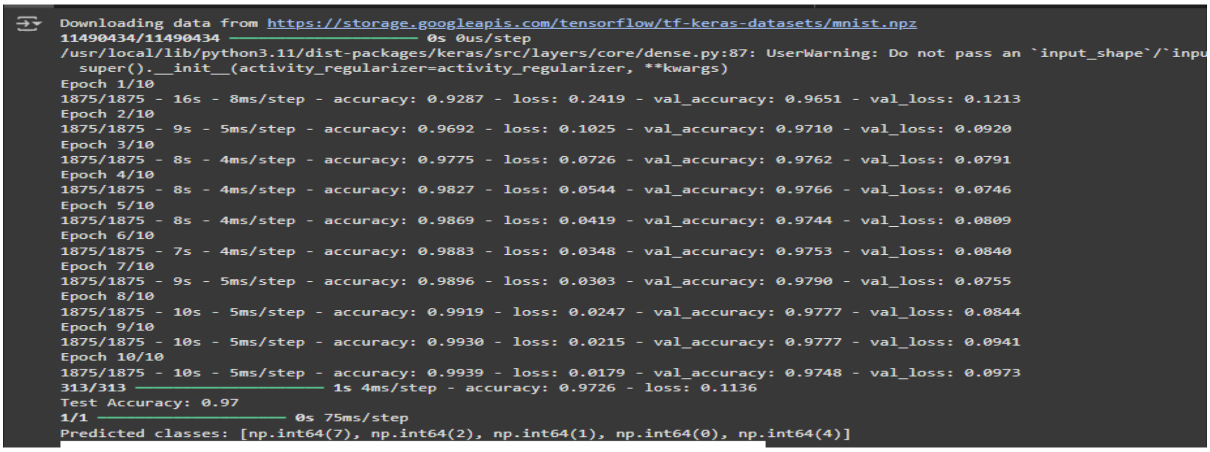
plt.xlabel('Epochs')

plt.ylabel('Accuracy')

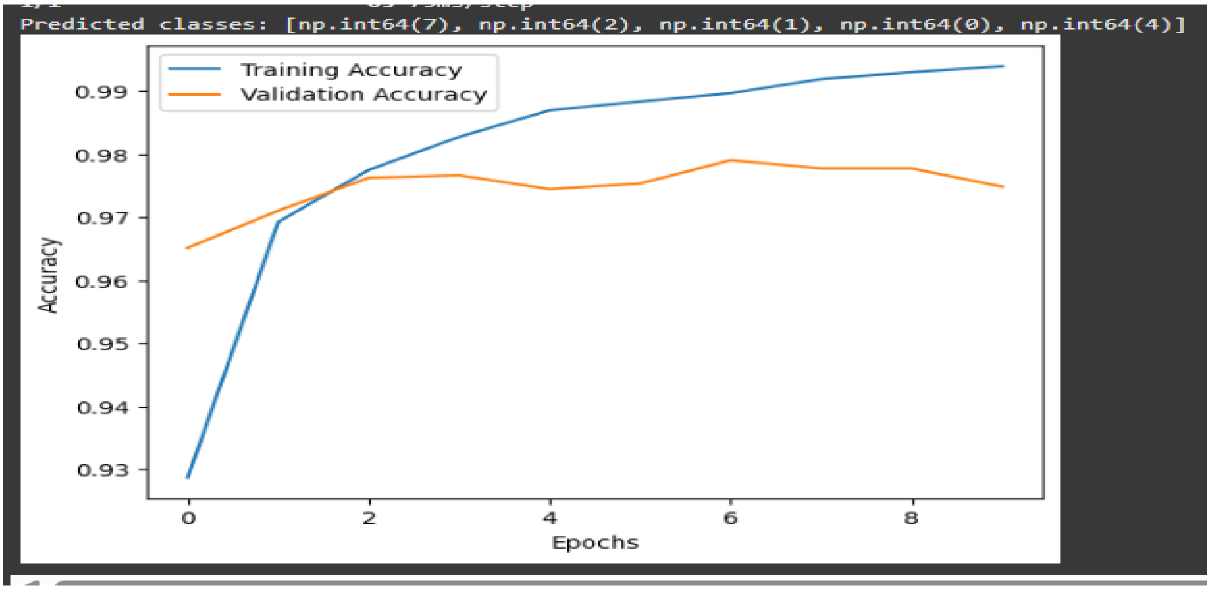
plt.legend()

plt.show()

Output :



```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/10
1875/1875 - 16s - 8ms/step - accuracy: 0.9287 - loss: 0.2419 - val_accuracy: 0.9651 - val_loss: 0.1213
Epoch 2/10
1875/1875 - 9s - 5ms/step - accuracy: 0.9692 - loss: 0.1025 - val_accuracy: 0.9710 - val_loss: 0.0920
Epoch 3/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9775 - loss: 0.0726 - val_accuracy: 0.9762 - val_loss: 0.0791
Epoch 4/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9827 - loss: 0.0544 - val_accuracy: 0.9766 - val_loss: 0.0746
Epoch 5/10
1875/1875 - 8s - 4ms/step - accuracy: 0.9869 - loss: 0.0419 - val_accuracy: 0.9744 - val_loss: 0.0809
Epoch 6/10
1875/1875 - 7s - 4ms/step - accuracy: 0.9883 - loss: 0.0348 - val_accuracy: 0.9753 - val_loss: 0.0840
Epoch 7/10
1875/1875 - 9s - 5ms/step - accuracy: 0.9896 - loss: 0.0303 - val_accuracy: 0.9790 - val_loss: 0.0755
Epoch 8/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9919 - loss: 0.0247 - val_accuracy: 0.9777 - val_loss: 0.0844
Epoch 9/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9930 - loss: 0.0215 - val_accuracy: 0.9777 - val_loss: 0.0941
Epoch 10/10
1875/1875 - 10s - 5ms/step - accuracy: 0.9939 - loss: 0.0179 - val_accuracy: 0.9748 - val_loss: 0.0973
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.9726 - loss: 0.1136
Test Accuracy: 0.97
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 75ms/step
Predicted classes: [np.int64(7), np.int64(2), np.int64(1), np.int64(0), np.int64(4)]
```



Practical 4

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

Code :

```
import tensorflow as tf

from tensorflow.keras.models import Model

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

import numpy as np

import matplotlib.pyplot as plt


# Generate a small synthetic dataset

X_train = np.random.rand(10, 128, 128, 3)

y_train = np.random.randint(0, 2, (10, 128, 128, 1))

X_test = np.random.rand(2, 128, 128, 3)

y_test = np.random.randint(0, 2, (2, 128, 128, 1))


# Define a simple U-Net model

def build_simple_unet():

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


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

    p1 = MaxPooling2D((2, 2))(c1)


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

    p2 = MaxPooling2D((2, 2))(c2)


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

```
    u1 = UpSampling2D((2, 2))(c3)
    u1 = concatenate([u1, c2])
    c4 = Conv2D(32, (3, 3), activation='relu', padding='same')(u1)

    u2 = UpSampling2D((2, 2))(c4)
    u2 = concatenate([u2, c1])
    c5 = Conv2D(16, (3, 3), activation='relu', padding='same')(u2)

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

    model = Model(inputs, outputs)
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
    return model

model = build_simple_unet()
model.summary()

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=2, validation_data=(X_test,
y_test))

# Test on a new image
def visualize_segmentation(model, image):
    pred_mask = model.predict(np.expand_dims(image, axis=0))[0]
    pred_mask = (pred_mask > 0.5).astype(np.uint8)

    plt.figure(figsize=(10, 5))
    plt.subplot(1, 2, 1)
    plt.title('Original Image')
    plt.imshow(image)
```

plt.subplot(1, 2, 2)

plt.title('Predicted Mask')

plt.imshow(pred_mask.squeeze(), cmap='gray')

plt.show()

 # Test on a random image

random_idx = np.random.randint(0, X_test.shape[0])

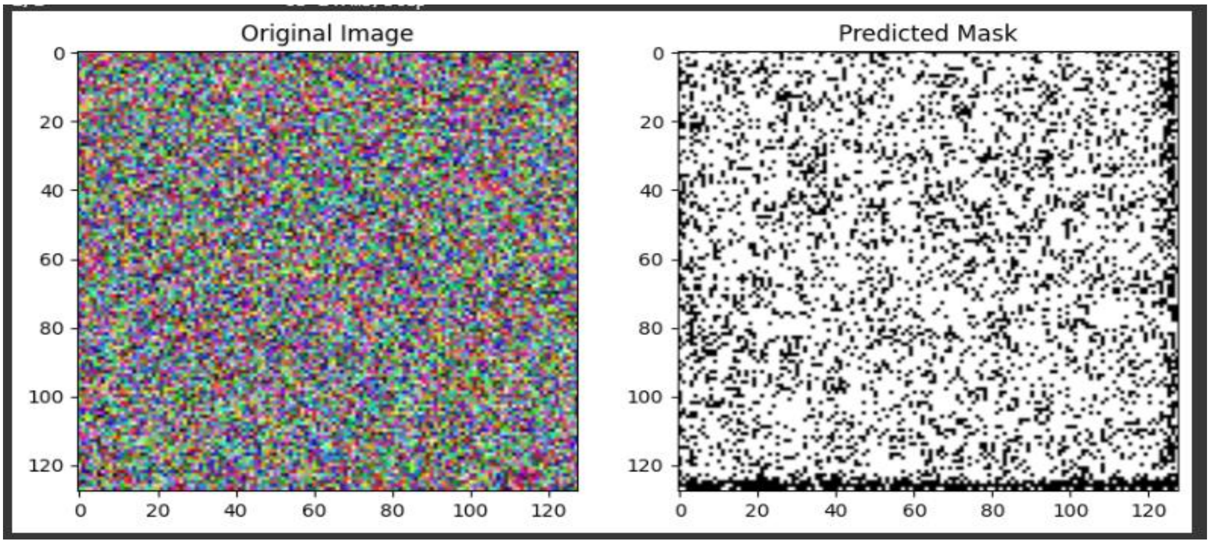
visualize_segmentation(model, X_test[random_idx])

Output :

Model: "functional"

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 128, 128, 3)	0	-
conv2d (Conv2D)	(None, 128, 128, 16)	448	input_layer[0][0]
max_pooling2d (MaxPooling2D)	(None, 64, 64, 16)	0	conv2d[0][0]
conv2d_1 (Conv2D)	(None, 64, 64, 32)	4,640	max_pooling2d[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 32, 32, 32)	0	conv2d_1[0][0]
conv2d_2 (Conv2D)	(None, 32, 32, 64)	18,496	max_pooling2d_1[0][0]
up_sampling2d (UpSampling2D)	(None, 64, 64, 64)	0	conv2d_2[0][0]
concatenate (Concatenate)	(None, 64, 64, 96)	0	up_sampling2d[0][0], conv2d_1[0][0]
conv2d_3 (Conv2D)	(None, 64, 64, 32)	27,680	concatenate[0][0]
up_sampling2d_1 (UpSampling2D)	(None, 128, 128, 32)	0	conv2d_3[0][0]
concatenate_1 (Concatenate)	(None, 128, 128, 48)	0	up_sampling2d_1[0][0], conv2d[0][0]
conv2d_4 (Conv2D)	(None, 128, 128, 16)	6,928	concatenate_1[0][0]
conv2d_5 (Conv2D)	(None, 128, 128, 1)	17	conv2d_4[0][0]

Total params: 58,209 (227.38 KB)
Trainable params: 58,209 (227.38 KB)
Non-trainable params: 0 (0.00 B)



Practical 5

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

Code :

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, LSTM, Embedding, Dropout, add
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.applications.vgg16 import VGG16, preprocess_input
import tensorflow as tf

# Define a small dataset
images = ["/content/Dog.jpg", "/content/cat.jpg", "/content/bike.jpg"]
captions = [
    "startseq a dog running in the park endseq",
    "startseq a cat sitting on a couch endseq",
    "startseq a person riding a bike endseq"
]

# Load VGG16 model for image feature extraction
base_model = VGG16(weights='imagenet')
model_vgg = Model(inputs=base_model.input, outputs=base_model.layers[-2].output)

def extract_features(image_path):
    img = load_img(image_path, target_size=(224, 224))
    img_array = img_to_array(img)
```

```
    img_array = np.expand_dims(img_array, axis=0)
    img_array = preprocess_input(img_array)
    return model_vgg.predict(img_array)

# Tokenize the captions
tokenizer = Tokenizer()
tokenizer.fit_on_texts(captions)
max_length = max(len(c.split()) for c in captions)
vocab_size = len(tokenizer.word_index) + 1

# Convert captions to sequences
sequences = tokenizer.texts_to_sequences(captions)
padded_captions = pad_sequences(sequences, maxlen=max_length, padding='post')

# Define the model
embedding_dim = 256
hidden_units = 256
image_input = Input(shape=(4096,))
image_output = Dense(embedding_dim, activation='relu')(image_input)
caption_input = Input(shape=(max_length,))
embedding = Embedding(vocab_size, embedding_dim, mask_zero=True)(caption_input)
lstm_out = LSTM(hidden_units)(embedding)
combined = add([image_output, lstm_out])
decoder_output = Dense(vocab_size, activation='softmax')(combined)
model = Model(inputs=[image_input, caption_input], outputs=decoder_output)
model.compile(loss='categorical_crossentropy', optimizer='adam')

# Dummy training labels
y_train = np.zeros((3, vocab_size))
model.fit([np.random.rand(3, 4096), padded_captions], y_train, epochs=5, batch_size=1)
```

```
# Generate a caption
def generate_caption(model, tokenizer, image_feature, max_length):
    in_text = 'startseq'
    for _ in range(max_length):
        sequence = tokenizer.texts_to_sequences([in_text])[0]
        sequence = pad_sequences([sequence], maxlen=max_length)
        yhat = np.argmax(model.predict([image_feature, sequence], verbose=0))
        word = tokenizer.index_word.get(yhat, None)
        if word is None:
            break
        in_text += ' ' + word
        if word == 'endseq':
            break
    return in_text.replace('startseq', '').replace('endseq', '').strip()

# Test on a sample image
sample_image_path = images[0]
plt.imshow(load_img(sample_image_path))
sample_feature = extract_features(sample_image_path)
caption = generate_caption(model, tokenizer, sample_feature, max_length)
print("Predicted Caption:", caption)
```

Output :

Practical 6

Applying the Autoencoder algorithms for encoding real-world data

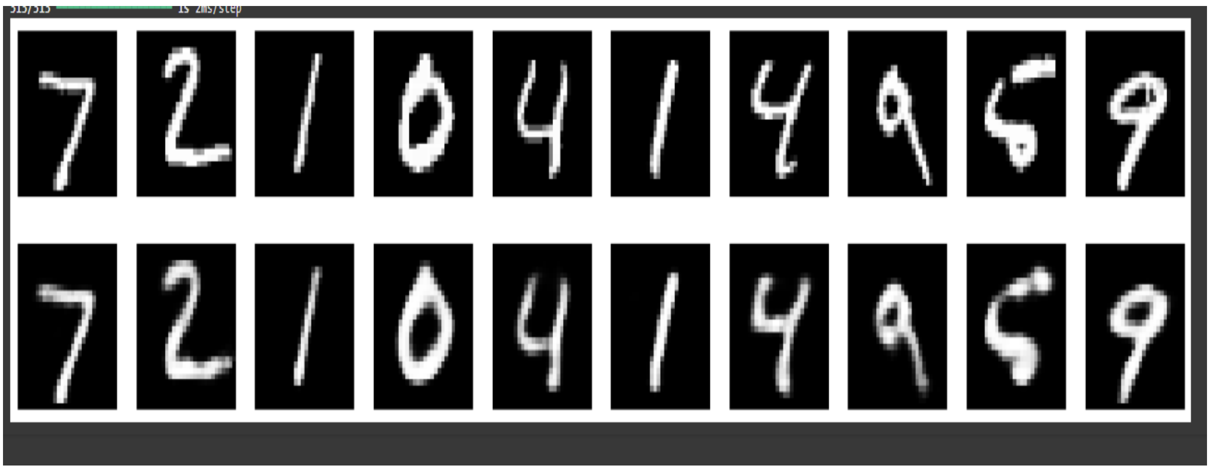
Code :

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
from tensorflow.keras.datasets import mnist

# Load and preprocess the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))
x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))

# Define the Autoencoder architecture
input_img = Input(shape=(28, 28, 1))

# Encoder
x = Flatten()(input_img)
x = Dense(128, activation='relu')(x)
x = Dense(64, activation='relu')(x)
encoded = Dense(32, activation='relu')(x)

# Decoder
x = Dense(64, activation='relu')(encoded)
```

```
x = Dense(128, activation='relu')(x)
x = Dense(28 * 28, activation='sigmoid')(x)
decoded = Reshape((28, 28, 1))(x)

# Define the Autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the Autoencoder
autoencoder.fit(
    x_train, x_train,
    epochs=50,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test, x_test)
)

# Encode and decode some images
encoded_imgs = autoencoder.predict(x_test)

# Display original and reconstructed images
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
```

```
    # Reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.show()
```

Output :



Practical 7

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

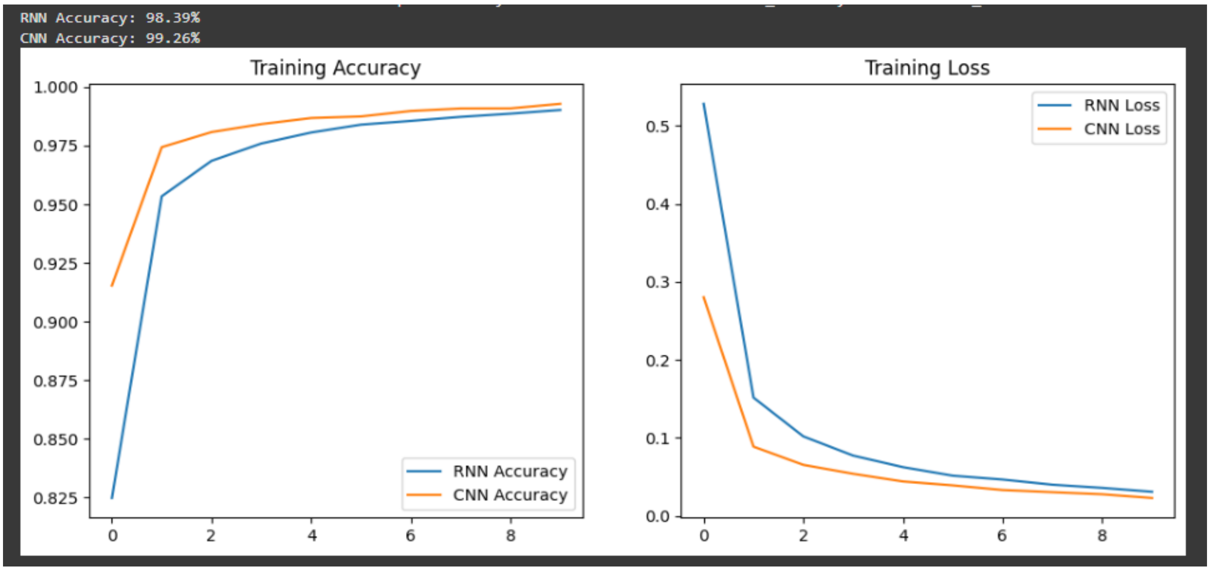
Code :

```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to_categorical


# Load and preprocess the MNIST dataset

(x_train, y_train), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.0

x_test = x_test.astype('float32') / 255.0


# Reshape data for CNN

x_train_cnn = x_train.reshape(-1, 28, 28, 1)

x_test_cnn = x_test.reshape(-1, 28, 28, 1)

# Reshape data for RNN

x_train_rnn = x_train.reshape(-1, 28, 28)

x_test_rnn = x_test.reshape(-1, 28, 28)


# One-hot encode the labels

y_train = to_categorical(y_train, 10)

y_test = to_categorical(y_test, 10)
```

```
# Define RNN model
rnn_model = Sequential([
    LSTM(128, input_shape=(28, 28)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')])

rnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Define CNN model
cnn_model = Sequential([
    Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Conv2D(64, kernel_size=(3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')])

cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Train the RNN model
print("Training RNN model...")
rnn_history = rnn_model.fit(x_train_rnn, y_train, epochs=10, batch_size=128,
validation_data=(x_test_rnn, y_test))
# Train the CNN model
print("Training CNN model...")
cnn_history = cnn_model.fit(x_train_cnn, y_train, epochs=10, batch_size=128,
validation_data=(x_test_cnn, y_test))

# Evaluate the models
```

```
rnn_score = rnn_model.evaluate(x_test_rnn, y_test, verbose=0)
cnn_score = cnn_model.evaluate(x_test_cnn, y_test, verbose=0)

print(f"RNN Accuracy: {rnn_score[1] * 100:.2f}%")
print(f"CNN Accuracy: {cnn_score[1] * 100:.2f}%")

# Plot training loss and accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(rnn_history.history['accuracy'], label='RNN Accuracy')
plt.plot(cnn_history.history['accuracy'], label='CNN Accuracy')
plt.legend()
plt.title('Training Accuracy')

plt.subplot(1, 2, 2)
plt.plot(rnn_history.history['loss'], label='RNN Loss')
plt.plot(cnn_history.history['loss'], label='CNN Loss')
plt.legend()
plt.title('Training Loss')
plt.show()
```

Output :



Practical 8

Write a program to develop Autoencoders using MNIST Handwritten Digits

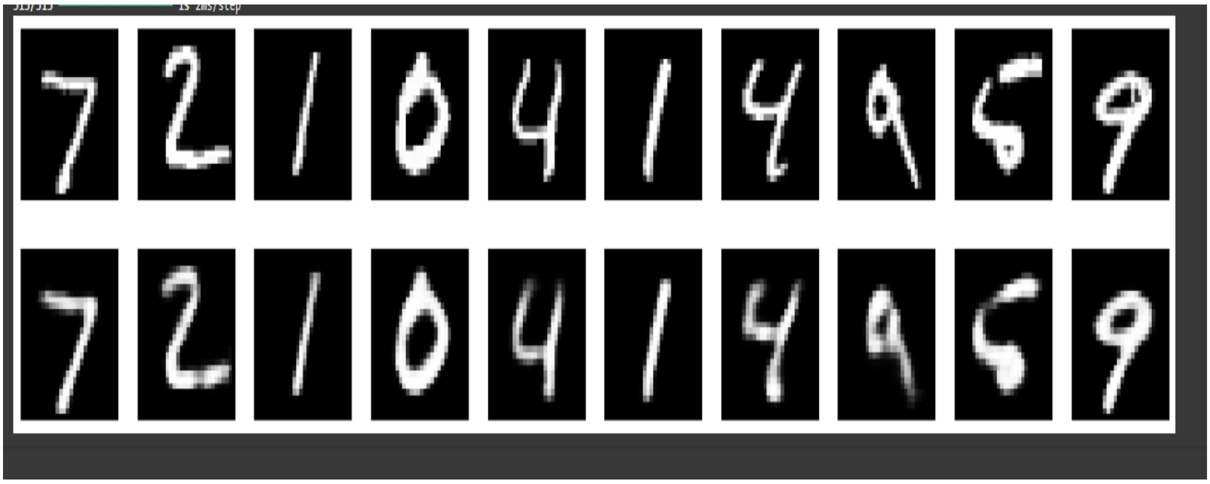
Code :

```
import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Model

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

from tensorflow.keras.datasets import mnist


# Load and preprocess the MNIST dataset

(x_train, _), (x_test, _) = mnist.load_data()

x_train = x_train.astype('float32') / 255.

x_test = x_test.astype('float32') / 255.

x_train = np.reshape(x_train, (len(x_train), 28, 28, 1))

x_test = np.reshape(x_test, (len(x_test), 28, 28, 1))


# Define the Autoencoder architecture

input_img = Input(shape=(28, 28, 1))


# Encoder

x = Flatten()(input_img)

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

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

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


# Decoder
```

```
x = Dense(64, activation='relu')(encoded)
x = Dense(128, activation='relu')(x)
x = Dense(28 * 28, activation='sigmoid')(x)
decoded = Reshape((28, 28, 1))(x)

# Define the Autoencoder model
autoencoder = Model(input_img, decoded)

# Compile the model
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the Autoencoder
autoencoder.fit(
    x_train, x_train,
    epochs=50,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test, x_test))

# Encode and decode some images
encoded_imgs = autoencoder.predict(x_test)

# Display original and reconstructed images
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
```

```
    # Reconstructed
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_imgs[i].reshape(28, 28), cmap='gray')
    plt.axis('off')
plt.show()
```

Output :



Practical 9

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

Code :

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
from sklearn.preprocessing import MinMaxScaler

# Load Google stock price data
import kagglehub
from kagglehub import KaggleDatasetAdapter

# Set the path to the file you'd like to load
file_path = "Google_Stock_Price_Train.csv"

# Load the latest version
df = kagglehub.load_dataset(
  KaggleDatasetAdapter.PANDAS,
  "vaibhavsxn/google-stock-prices-training-and-test-data",
  file_path,
  # Provide any additional arguments like
  # sql_query or pandas_kwargs. See the
  # documenation for more information:
```

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

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

# Extract the 'Close' prices and preprocess
stock_prices = df['Close'].str.replace(',',",regex=True).astype(float).values
stock_prices = stock_prices.reshape(-1, 1)
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_prices = scaler.fit_transform(stock_prices)

# Prepare data for RNN
sequence_length = 60
X, y = [], []
for i in range(len(scaled_prices) - sequence_length):
    X.append(scaled_prices[i:i+sequence_length])
    y.append(scaled_prices[i+sequence_length])
X, y = np.array(X), np.array(y)

# Reshape X for LSTM input
X = np.reshape(X, (X.shape[0], X.shape[1], 1))

# Build RNN model
model = Sequential([
    LSTM(units=50, return_sequences=True, input_shape=(X.shape[1], 1)),
    Dropout(0.2),
    LSTM(units=50, return_sequences=False),
    Dropout(0.2),
    Dense(units=1)])

# Compile and train the model
```

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

model.fit(X, y, epochs=50, batch_size=32)

 # Make predictions

predicted_prices = model.predict(X)

predicted_prices = scaler.inverse_transform(predicted_prices)

 # Plot actual vs predicted stock prices

plt.plot(stock_prices[sequence_length:], color='blue', label='Actual Google Stock Price')

plt.plot(predicted_prices, color='red', label='Predicted Google Stock Price')

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

plt.xlabel('Time')

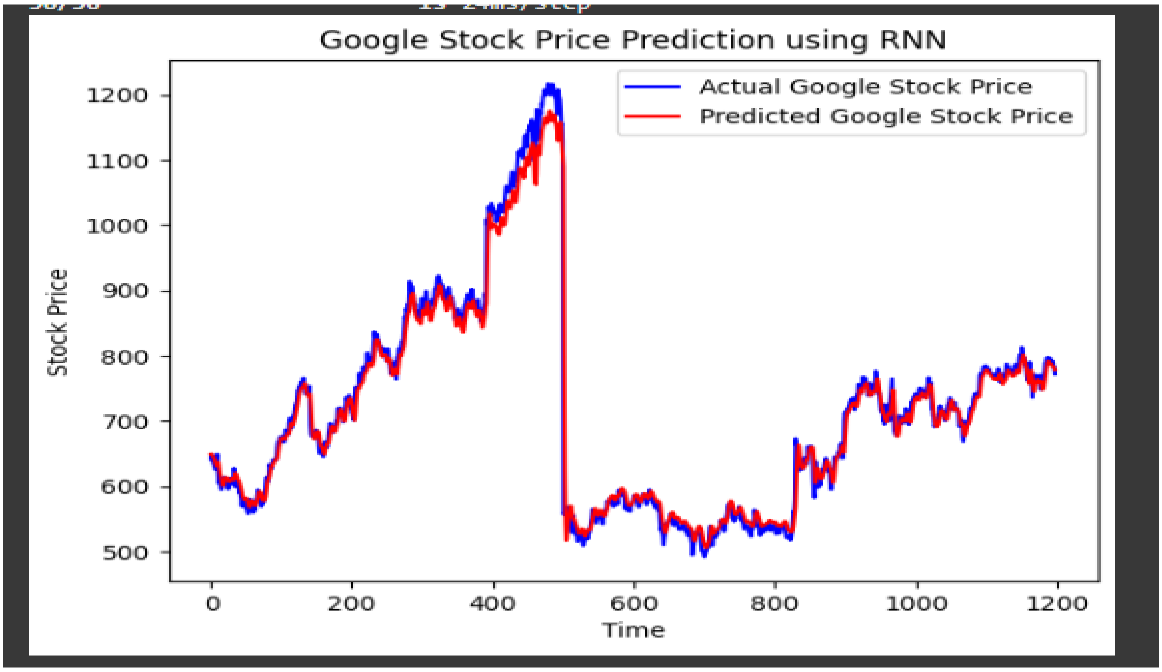
plt.ylabel('Stock Price')

plt.legend()

plt.show()

Output :

```
<ipython-input-3-6e5fd158cc1f>:17: DeprecationWarning: load_dataset is deprecated and will be removed in future version.
  df = kagglehub.load_dataset(
First 5 records:        Date    Open    High     Low   Close      Volume
0  1/3/2012  325.25  332.83  324.97  663.59   7,380,500
1  1/4/2012  331.27  333.87  329.08  666.45   5,749,400
2  1/5/2012  329.83  330.75  326.89  657.21   6,590,300
3  1/6/2012  328.34  328.77  323.68  648.24   5,405,900
4  1/9/2012  322.04  322.29  309.46  620.76  11,688,800
```



Practical 10

Applying Generative Adversarial Networks for image generation and unsupervised tasks.

Code :

```
import tensorflow as tf

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

from tensorflow.keras.models import Sequential

import numpy as np

import matplotlib.pyplot as plt


# Define the Generator

def build_generator():

    model = Sequential()

    model.add(Dense(7 * 7 * 256, input_dim=100))

    model.add(LeakyReLU(0.2))

    model.add(Reshape((7, 7, 256)))


    model.add(Conv2DTranspose(128, kernel_size=4, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(0.2))


    model.add(Conv2DTranspose(64, kernel_size=4, strides=2, padding='same'))

    model.add(BatchNormalization())

    model.add(LeakyReLU(0.2))


    model.add(Conv2D(1, kernel_size=7, padding='same', activation='tanh'))

    return model
```

Define the Discriminator

```
def build_discriminator():
    model = Sequential()
    model.add(Conv2D(64, kernel_size=3, strides=2, padding='same', input_shape=(28, 28,
1)))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.3))

    model.add(Conv2D(128, kernel_size=3, strides=2, padding='same'))
    model.add(LeakyReLU(0.2))
    model.add(Dropout(0.3))

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

Compile the GAN

```
def build_gan(generator, discriminator):
    discriminator.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    discriminator.trainable = False
    model = Sequential([generator, discriminator])
    model.compile(loss='binary_crossentropy', optimizer='adam')
    return model
```

Load dataset (MNIST for simplicity)

```
(X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
X_train = X_train / 127.5 - 1.0
X_train = np.expand_dims(X_train, axis=-1)
```

Training loop

```python
def train_gan(gan, generator, discriminator, epochs=100, batch_size=64, sample_interval=1000):
    valid = np.ones((batch_size, 1))
    fake = np.zeros((batch_size, 1))

    for epoch in range(epochs):
        idx = np.random.randint(0, X_train.shape[0], batch_size)
        real_imgs = X_train[idx]

        noise = np.random.normal(0, 1, (batch_size, 100))
        fake_imgs = generator.predict(noise)

        d_loss_real = discriminator.train_on_batch(real_imgs, valid)
        d_loss_fake = discriminator.train_on_batch(fake_imgs, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        noise = np.random.normal(0, 1, (batch_size, 100))
        g_loss = gan.train_on_batch(noise, valid)

        if epoch % sample_interval == 0:
            print(f"Epoch {epoch}, D Loss: {d_loss[0]}, G Loss: {g_loss}")
            sample_images(generator)

# Generate and display images
def sample_images(generator, image_grid_rows=4, image_grid_columns=4):
    noise = np.random.normal(0, 1, (image_grid_rows * image_grid_columns, 100))
    gen_imgs = generator.predict(noise)
    gen_imgs = 0.5 * gen_imgs + 0.5

    fig, axs = plt.subplots(image_grid_rows, image_grid_columns, figsize=(4, 4), sharex=True, sharey=True)
```

```
    count = 0
    for i in range(image_grid_rows):
        for j in range(image_grid_columns):
            axs[i, j].imshow(gen_imgs[count, :, :, 0], cmap='gray')
            axs[i, j].axis('off')
            count += 1
    plt.show()

# Build and compile the GAN
generator = build_generator()
discriminator = build_discriminator()
gan = build_gan(generator, discriminator)

# Train the GAN
train_gan(gan, generator, discriminator)
```

Output :