INDEX

Sr. No.		Practical	Signature
1.	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 Tensor Flow	
	b.	Program to solve the XOR problem.	
2.	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. • Make predictions on new data points.	
3.	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	
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.	

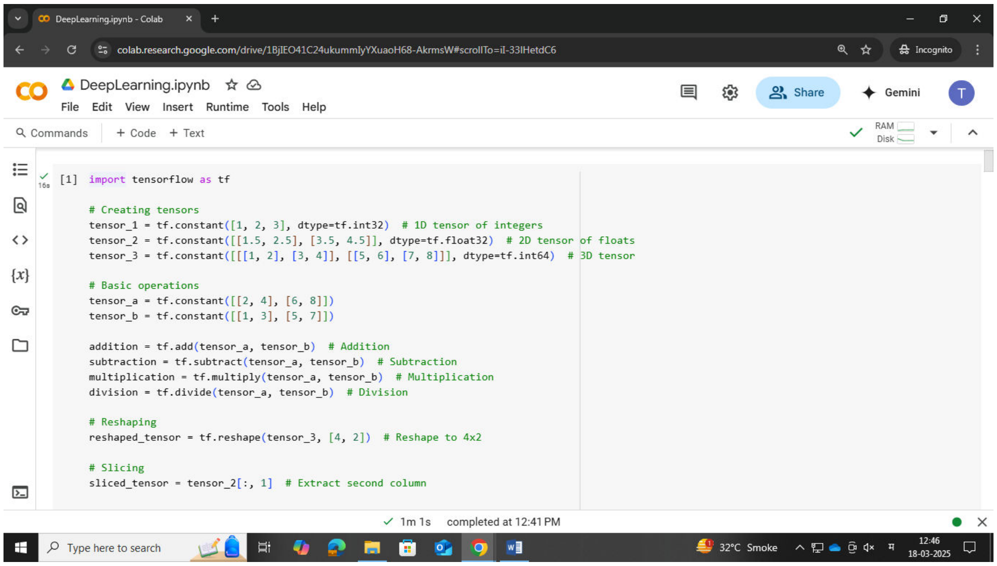
PRACTICAL 1 A.

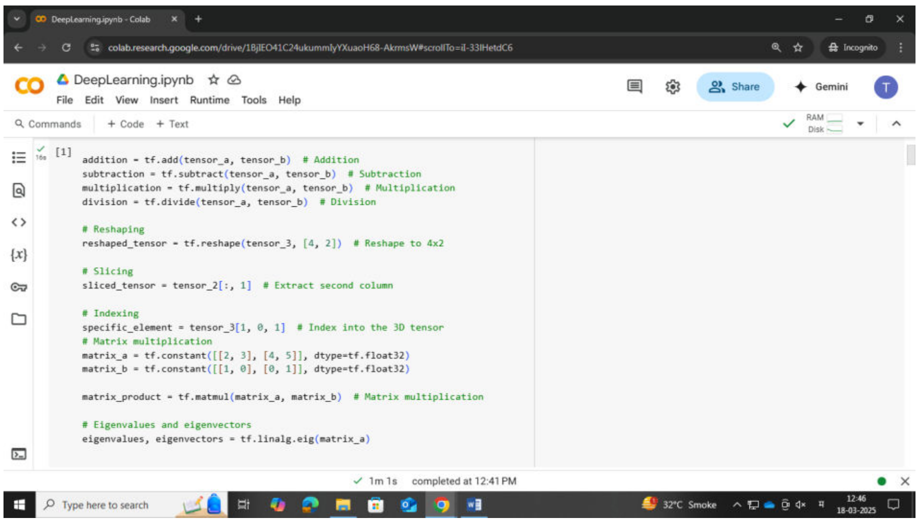
Aim:

- 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 Tensor Flow

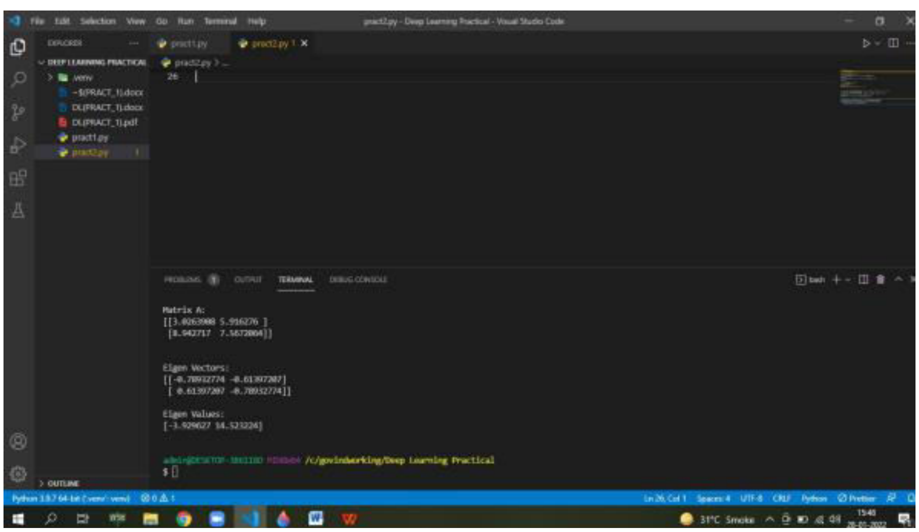
Code:

```
import tensorflow as tf
# Creating tensors
tensor_1 = tf.constant([1, 2, 3], dtype=tf.int32)  # 1D tensor of integers
tensor_2 = tf.constant([[1.5, 2.5], [3.5, 4.5]], dtype=tf.float32)  # 2D
tensor of floats
tensor_3 = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]],
dtype=tf.int64)  # 3D tensor

# Basic operations
tensor_a = tf.constant([[2, 4], [6, 8]])
tensor_b = tf.constant([[1, 3], [5, 7]])

addition = tf.add(tensor_a, tensor_b)  # Addition
subtraction = tf.subtract(tensor_a, tensor_b)  # Subtraction
multiplication = tf.multiply(tensor_a, tensor_b)  # Multiplication
division = tf.divide(tensor_a, tensor_b)  # Division
# Reshaping
reshaped_tensor = tf.reshape(tensor_3, [4, 2])  # Reshape to 4x2
# Slicing
sliced_tensor = tensor_2[:, 1]  # Extract second column

# Indexing
specific_element = tensor_3[1, 0, 1]  # Index into the 3D tensor
# Matrix multiplication
matrix_a = tf.constant([[2, 3], [4, 5]], dtype=tf.float32)
matrix_b = tf.constant([[1, 0], [0, 1]], dtype=tf.float32)

matrix_product = tf.matmul(matrix_a, matrix_b)  # Matrix multiplication

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = tf.linalg.eig(matrix_a)
```


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=iI-33lHetdC6
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands + Code + Text
[1] import tensorflow as tf
# Creating tensors
tensor_1 = tf.constant([1, 2, 3], dtype=tf.int32) # 1D tensor of integers
tensor_2 = tf.constant([[1.5, 2.5], [3.5, 4.5]], dtype=tf.float32) # 2D tensor of floats
tensor_3 = tf.constant([[[1, 2], [3, 4]], [[5, 6], [7, 8]]], dtype=tf.int64) # 3D tensor
# Basic operations
tensor_a = tf.constant([[2, 4], [6, 8]])
tensor_b = tf.constant([[1, 3], [5, 7]])
addition = tf.add(tensor_a, tensor_b) # Addition
subtraction = tf.subtract(tensor_a, tensor_b) # Subtraction
multiplication = tf.multiply(tensor_a, tensor_b) # Multiplication
division = tf.divide(tensor_a, tensor_b) # Division
# Reshaping
reshaped_tensor = tf.reshape(tensor_3, [4, 2]) # Reshape to 4x2
# Slicing
sliced_tensor = tensor_2[:, 1] # Extract second column
1m 1s completed at 12:41 PM
Type here to search
32°C Smoke
12:46
18-03-2025


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=iI-33lHetdC6
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Share
Gemini
Commands + Code + Text
[1]
addition = tf.add(tensor_a, tensor_b) # Addition
subtraction = tf.subtract(tensor_a, tensor_b) # Subtraction
multiplication = tf.multiply(tensor_a, tensor_b) # Multiplication
division = tf.divide(tensor_a, tensor_b) # Division
# Reshaping
reshaped_tensor = tf.reshape(tensor_3, [4, 2]) # Reshape to 4x2
# Slicing
sliced_tensor = tensor_2[:, 1] # Extract second column
# Indexing
specific_element = tensor_3[1, 0, 1] # Index into the 3D tensor
# Matrix multiplication
matrix_a = tf.constant([[2, 3], [4, 5]], dtype=tf.float32)
matrix_b = tf.constant([[1, 0], [0, 1]], dtype=tf.float32)
matrix_product = tf.matmul(matrix_a, matrix_b) # Matrix multiplication
# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = tf.linalg.eig(matrix_a)
1m 1s completed at 12:41 PM
Type here to search
32°C Smoke
12:46
18-03-2025

Output:

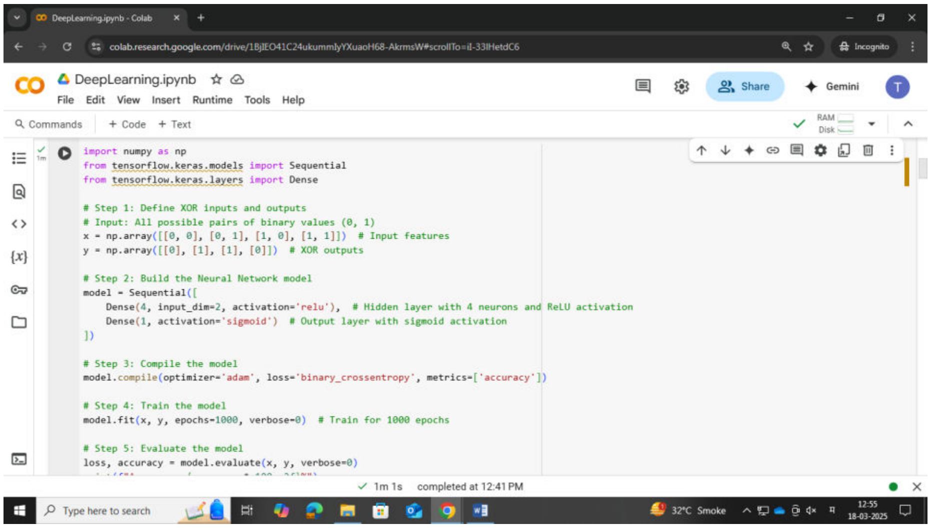

TERMINAL
Matrix A:
Eigen Vectors:
Eigen Values:
/c/govindworking/Deep Learning Practical

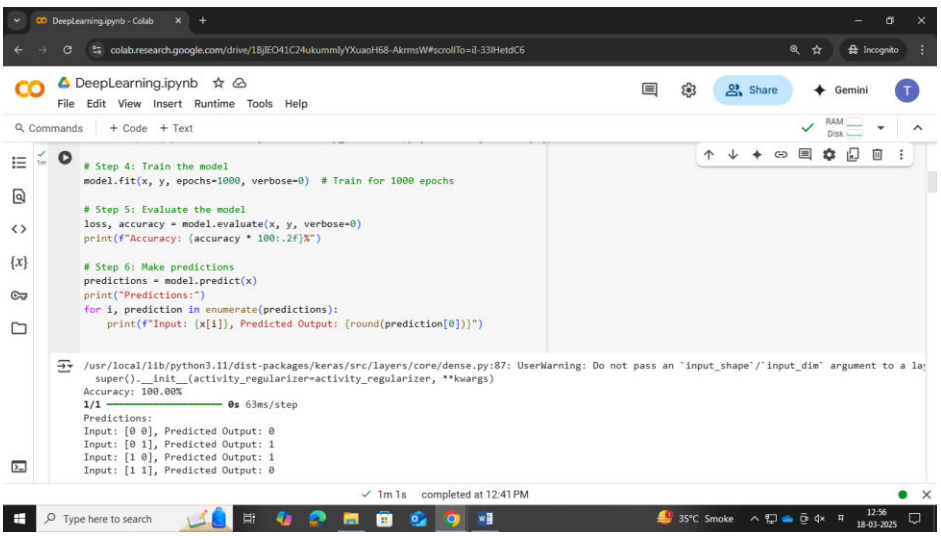
PRACTICAL 1 B.

Aim: Program to solve the XOR problem.

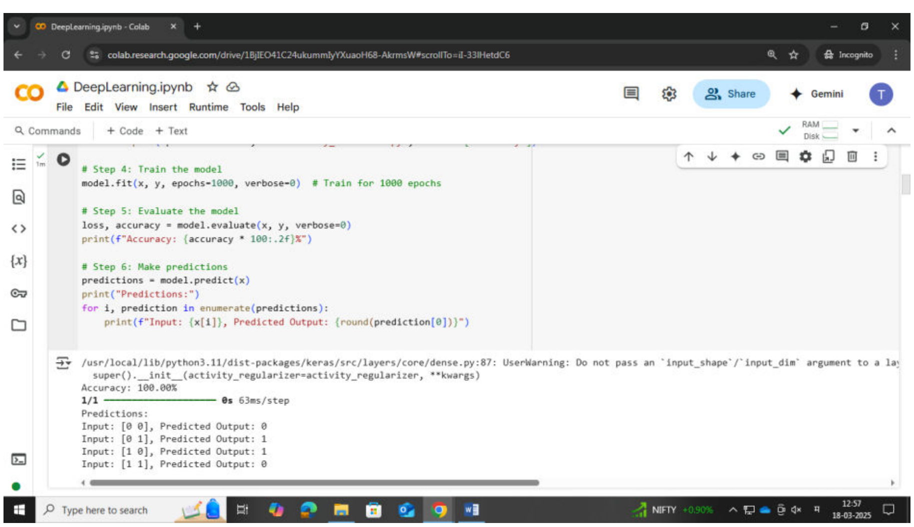
Code:

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Step 1: Define XOR inputs and outputs
# Input: All possible pairs of binary values (0, 1)
x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input features
y = np.array([[0], [1], [1], [0]])  # XOR outputs

# Step 2: Build the Neural Network model
model = Sequential([
    Dense(4, input_dim=2, activation='relu'),  # Hidden layer with 4
neurons and ReLU activation
    Dense(1, activation='sigmoid')  # Output layer with sigmoid activation
])

# Step 3: Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Step 4: Train the model
model.fit(x, y, epochs=1000, verbose=0)  # Train for 1000 epochs

# Step 5: Evaluate the model
loss, accuracy = model.evaluate(x, y, verbose=0)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Step 6: Make predictions
predictions = model.predict(x)
print("Predictions:")
for i, prediction in enumerate(predictions):
    print(f"Input: {x[i]}, Predicted Output: {round(prediction[0])}")
```


DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Step 1: Define XOR inputs and outputs
# Input: All possible pairs of binary values (0, 1)
x = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) # Input features
y = np.array([[0], [1], [1], [0]]) # XOR outputs
# Step 2: Build the Neural Network model
model = Sequential([
Dense(4, input_dim=2, activation='relu'), # Hidden layer with 4 neurons and ReLU activation
Dense(1, activation='sigmoid') # Output layer with sigmoid activation
])
# Step 3: Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
# Step 4: Train the model
model.fit(x, y, epochs=1000, verbose=0) # Train for 1000 epochs
Step 5: Evaluate the model
loss, accuracy = model.evaluate(x, y, verbose=0)
1m 1s completed at 12:41 PM


DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Step 4: Train the model
model.fit(x, y, epochs=1000, verbose=0) # Train for 1000 epochs
Step 5: Evaluate the model
loss, accuracy = model.evaluate(x, y, verbose=0)
print(f"Accuracy: {accuracy * 100:.2f}%")
# Step 6: Make predictions
predictions = model.predict(x)
print("Predictions:")
for i, prediction in enumerate(predictions):
print(f"Input: {x[i]}, Predicted Output: {round(prediction[0])}")
/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 lay
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Accuracy: 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
Predictions:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 0
1m 1s completed at 12:41 PM

Output:


DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Step 4: Train the model
model.fit(x, y, epochs=1000, verbose=0) # Train for 1000 epochs
Step 5: Evaluate the model
loss, accuracy = model.evaluate(x, y, verbose=0)
print(f"Accuracy: {accuracy * 100:.2f}%")
# Step 6: Make predictions
predictions = model.predict(x)
print("Predictions:")
for i, prediction in enumerate(predictions):
print(f"Input: {x[i]}, Predicted Output: {round(prediction[0])}")
/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 lay
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Accuracy: 100.00%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
Predictions:
Input: [0 0], Predicted Output: 0
Input: [0 1], Predicted Output: 1
Input: [1 0], Predicted Output: 1
Input: [1 1], Predicted Output: 0

PRACTICAL 2 A.

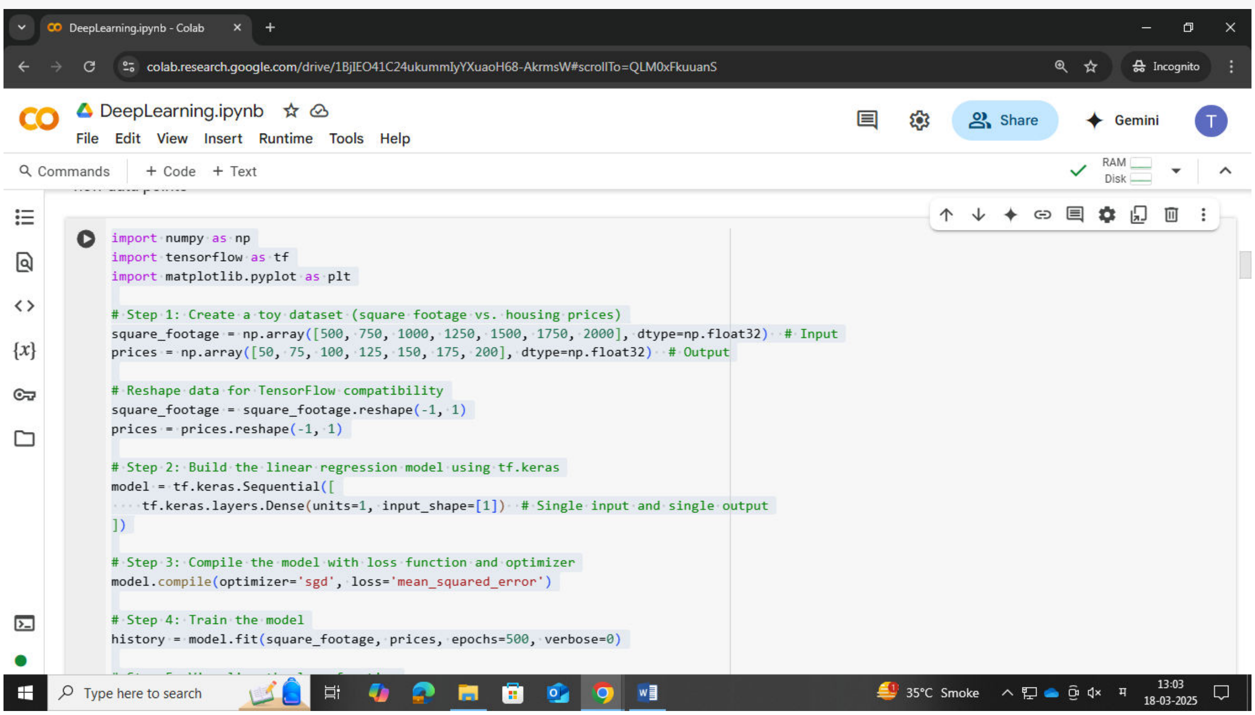
Aim:

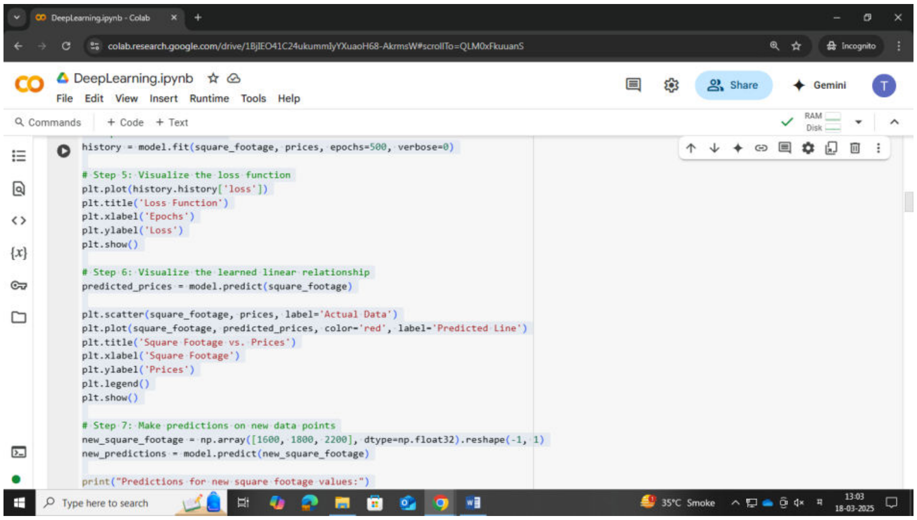
- Implement a simple linear regression model using TensorFlow's lowlevel API (or tf. keras).
- Train the model on a toy dataset (e.g., housing prices vs. square footage).
- Visualize the loss function and the learned linear relationship.
- Make predictions on new data points.

Code:

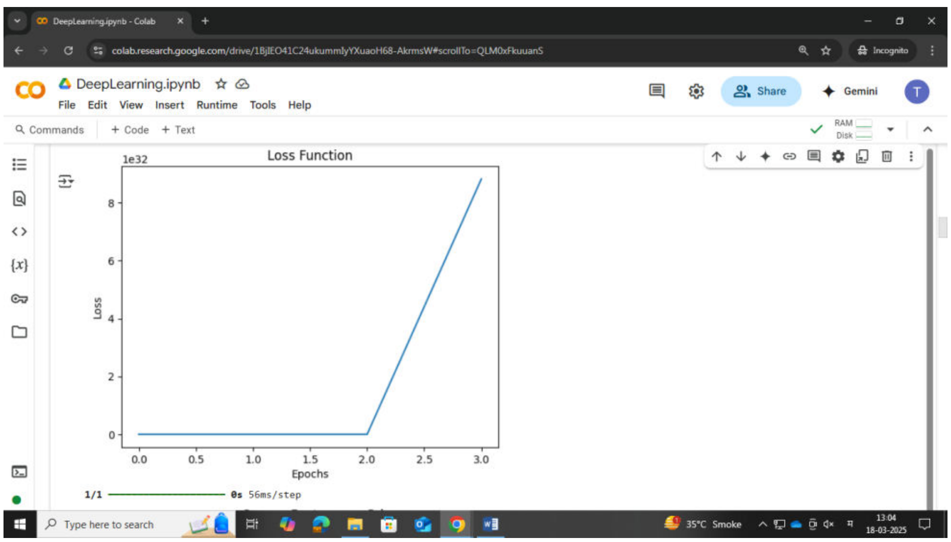
```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

# Step 1: Create a toy dataset (square footage vs. housing prices)
square_footage = np.array([500, 750, 1000, 1250, 1500, 1750, 2000],
dtype=np.float32)  # Input
prices = np.array([50, 75, 100, 125, 150, 175, 200], dtype=np.float32)  #
Output

# Reshape data for TensorFlow compatibility
square_footage = square_footage.reshape(-1, 1)
prices = prices.reshape(-1, 1)

# Step 2: Build the linear regression model using tf.keras
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, input_shape=[1])  # Single input and
single output
])

# Step 3: Compile the model with loss function and optimizer
model.compile(optimizer='sgd', loss='mean_squared_error')

# Step 4: Train the model
history = model.fit(square_footage, prices, epochs=500, verbose=0)

# Step 5: Visualize the loss function
plt.plot(history.history['loss'])
plt.title('Loss Function')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```

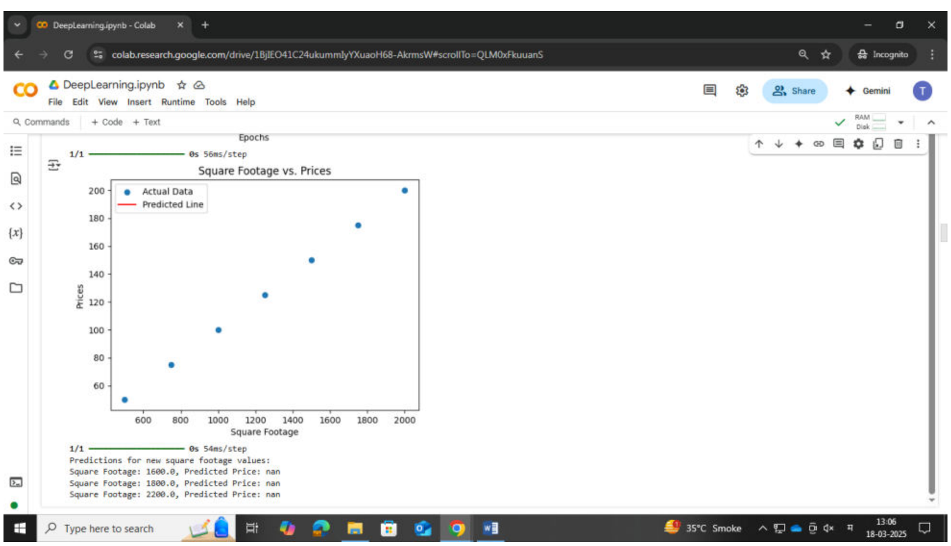
```
# Step 6: Visualize the learned linear relationship
predicted_prices = model.predict(square_footage)

plt.scatter(square_footage, prices, label='Actual Data')
plt.plot(square_footage, predicted_prices, color='red', label='Predicted
Line')
plt.title('Square Footage vs. Prices')
plt.xlabel('Square Footage')
plt.ylabel('Prices')
plt.legend()
plt.show()

# Step 7: Make predictions on new data points
new_square_footage = np.array([1600, 1800, 2200],
dtype=np.float32).reshape(-1, 1)
new_predictions = model.predict(new_square_footage)

print("Predictions for new square footage values:")
for i, sqft in enumerate(new_square_footage):
    print(f"Square Footage: {sqft[0]}, Predicted Price:
{new_predictions[i][0]:.2f}")
```




DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
history = model.fit(square_footage, prices, epochs=500, verbose=0)
# Step 5: Visualize the loss function
plt.plot(history.history['loss'])
plt.title('Loss Function')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
# Step 6: Visualize the learned linear relationship
predicted_prices = model.predict(square_footage)
plt.scatter(square_footage, prices, label='Actual Data')
plt.plot(square_footage, predicted_prices, color='red', label='Predicted Line')
plt.title('Square Footage vs. Prices')
plt.xlabel('Square Footage')
plt.ylabel('Prices')
plt.legend()
plt.show()
# Step 7: Make predictions on new data points
new_square_footage = np.array([1600, 1800, 2200], dtype=np.float32).reshape(-1, 1)
new_predictions = model.predict(new_square_footage)
print("Predictions for new square footage values:")

Output ;


DeepLearning.ipynb
Loss Function
1e32
Loss
Epochs
1/1 0s 56ms/step


DeepLearning.ipynb
1/1 0s 56ms/step
Square Footage vs. Prices
Actual Data
Predicted Line
Prices
Square Footage
1/1 0s 54ms/step
Predictions for new square footage values:
Square Footage: 1600.0, Predicted Price: nan
Square Footage: 1800.0, Predicted Price: nan
Square Footage: 2200.0, Predicted Price: nan

PRACTICAL 3 A.

Aim: Implementing deep neural network for performing binary classification task

Code:

```
import numpy as np

import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Step 1: Create a Toy Dataset
# Features (X) and Labels (y) for binary classification
np.random.seed(42)
X = np.random.rand(500, 2)  # 500 samples with 2 features
y = (X[:, 0] + X[:, 1] > 1).astype(int)  # Label: 1 if sum of features >
1, else 0

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 2: Build the Deep Neural Network
model = Sequential([
    Dense(16, input_dim=2, activation='relu'),  # Hidden layer with 16
neurons
    Dense(8, activation='relu'),  # Hidden layer with 8 neurons
    Dense(1, activation='sigmoid')  # Output layer with sigmoid activation
for binary classification
])

# Step 3: Compile the Model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Step 4: Train the Model
```

```
history = model.fit(X_train, y_train, epochs=50, batch_size=16,
validation_split=0.2, verbose=0)

# Step 5: Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# Step 6: Visualize the Training Process
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Step 7: Make Predictions on New Data
new_data = np.array([[0.2, 0.8], [0.6, 0.4]])
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)

print("Predictions on new data:")
for i, pred in enumerate(predictions):
    print(f"Input: {new_data[i]}, Predicted Class: {int(pred > 0.5)}")
```

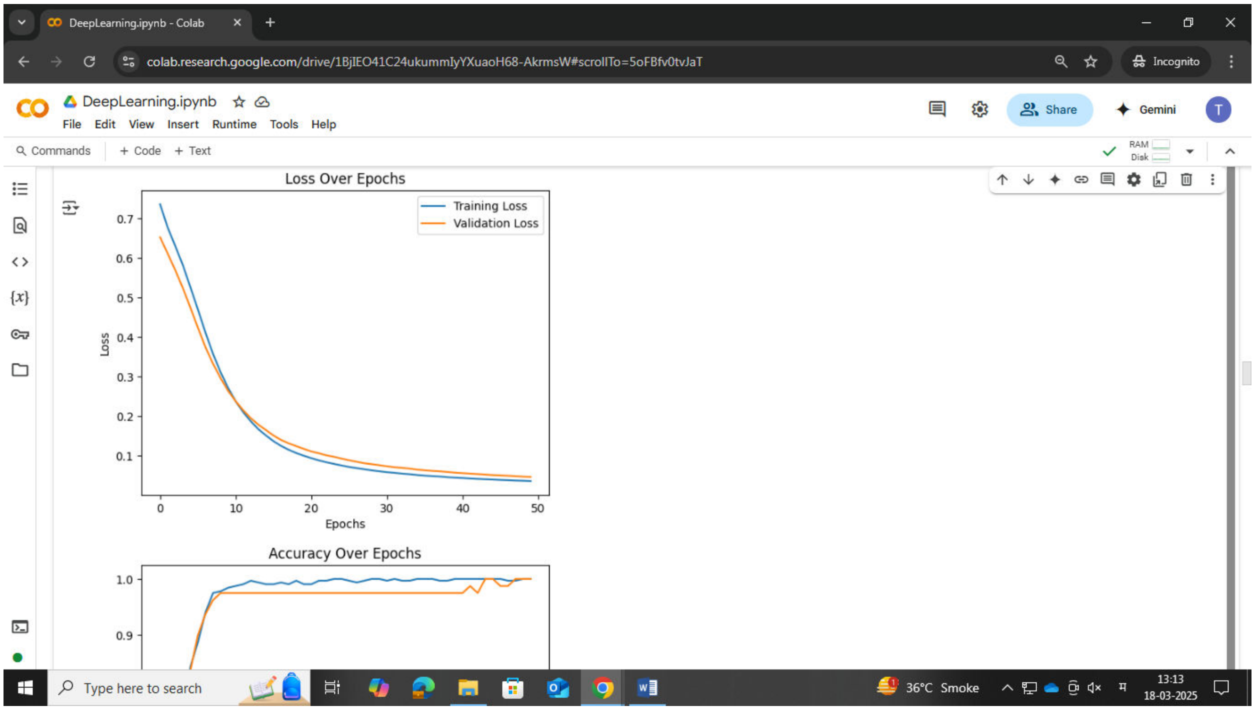
```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

# Step 1: Create a Toy Dataset
# Features (X) and Labels (y) for binary classification
np.random.seed(42)
X = np.random.rand(500, 2)  # 500 samples with 2 features
y = (X[:, 0] + X[:, 1] > 1).astype(int)  # Label: 1 if sum of features > 1, else 0

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 2: Build the Deep Neural Network
model = Sequential([
    Dense(16, input_dim=2, activation='relu'),  # Hidden layer with 16 neurons
    Dense(8, activation='relu'),  # Hidden layer with 8 neurons
    Dense(1, activation='sigmoid')  # Output layer with sigmoid activation for binary classification
])

# Step 3: Compile the Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
```

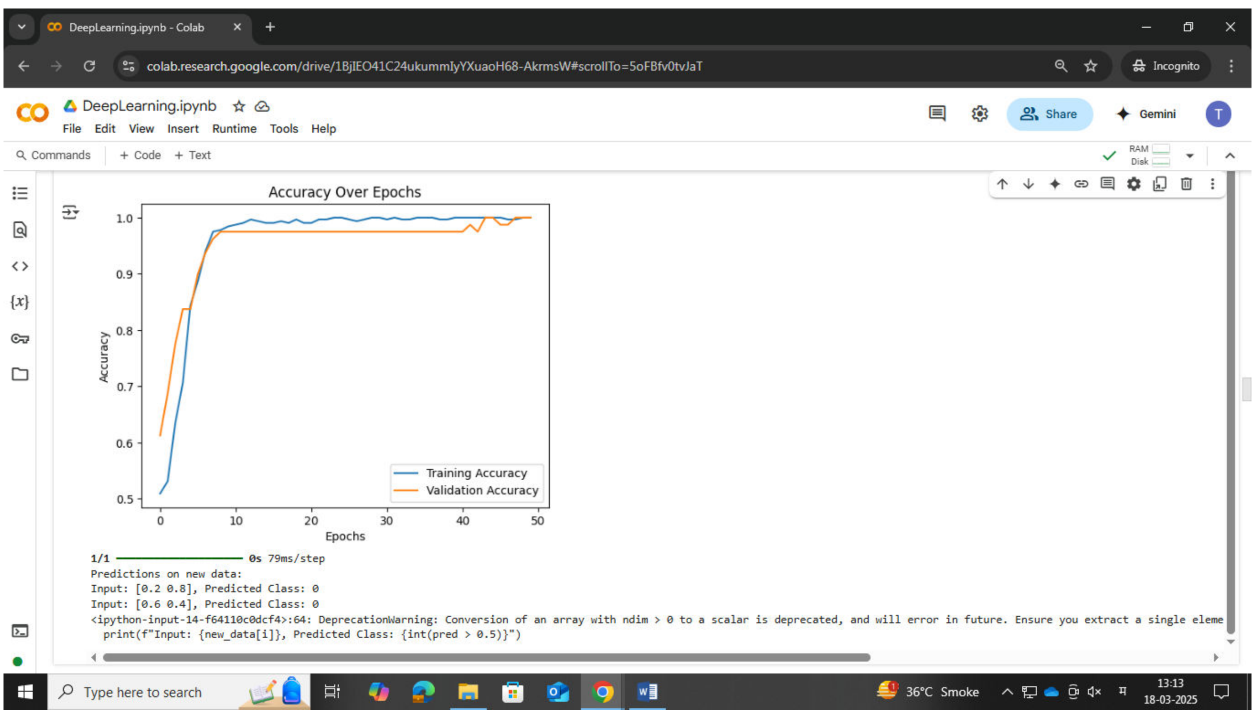
```python
# Step 3: Compile the Model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Step 4: Train the Model
history = model.fit(X_train, y_train, epochs=50, batch_size=16, validation_split=0.2, verbose=0)

# Step 5: Evaluate the Model
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {accuracy * 100:.2f}%")

# Step 6: Visualize the Training Process
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# Step 7: Make Predictions on New Data
new_data = np.array([[0.2, 0.8], [0.6, 0.4]])
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)
```

```python
# Step 7: Make Predictions on New Data
new_data = np.array([[0.2, 0.8], [0.6, 0.4]])
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)

print("Predictions on new data:")
for i, pred in enumerate(predictions):
    print(f"Input: {new_data[i]}, Predicted Class: {int(pred > 0.5)}")
```

```
/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,
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Test Accuracy: 100.00%
```

Output:


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=5oFBfv0tvJaT
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Loss Over Epochs
Training Loss
Validation Loss
Loss
Epochs
Accuracy Over Epochs
Type here to search
36°C Smoke
13:13
18-03-2025


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=5oFBfv0tvJaT
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Accuracy Over Epochs
Training Accuracy
Validation Accuracy
Accuracy
Epochs
1/1 0s 79ms/step
Predictions on new data:
Input: [0.2 0.8], Predicted Class: 0
Input: [0.6 0.4], Predicted Class: 0
<ipython-input-14-f64110c0dcf4>:64: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single eleme
print(f"Input: {new_data[i]}, Predicted Class: {int(pred > 0.5)}")
Type here to search
36°C Smoke
13:13
18-03-2025

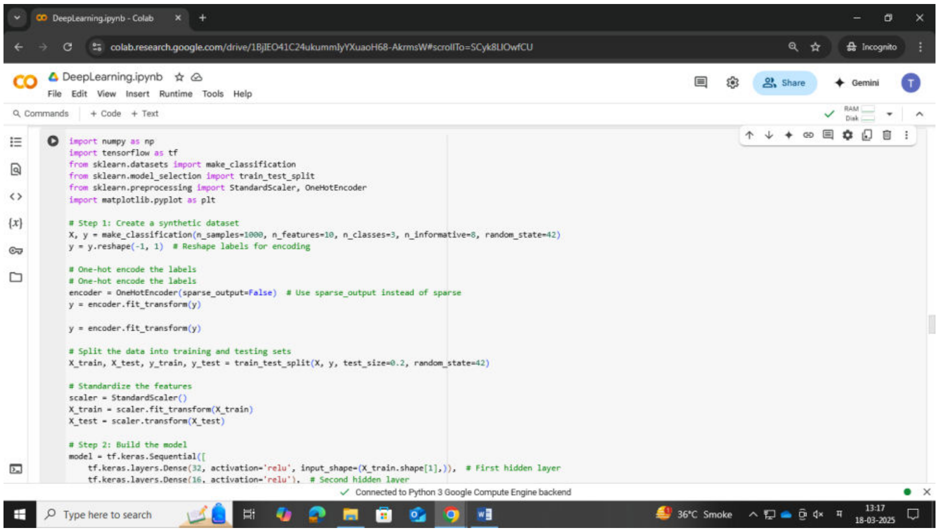
PRACTICAL 3 B.

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

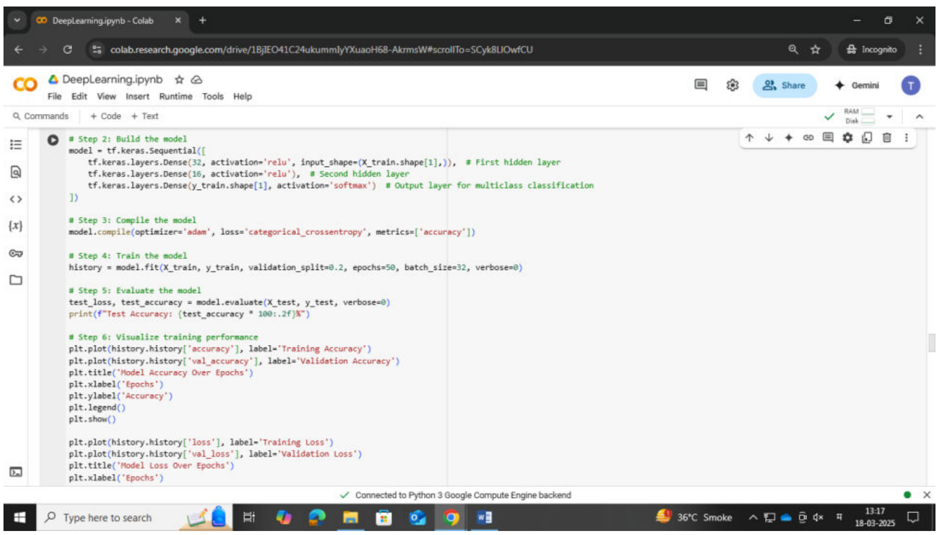
Code:

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

# Step 1: Create a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_classes=3,
n_informative=8, random_state=42)
y = y.reshape(-1, 1)  # Reshape labels for encoding

# One-hot encode the labels
# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)  # Use sparse_output instead
of sparse
y = encoder.fit_transform(y)

y = encoder.fit_transform(y)

# Split the data 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 2: Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu',
input_shape=(X_train.shape[1],)),  # First hidden layer
    tf.keras.layers.Dense(16, activation='relu'),  # Second hidden layer
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')  #
Output layer for multiclass classification
])

# Step 3: Compile the model
```

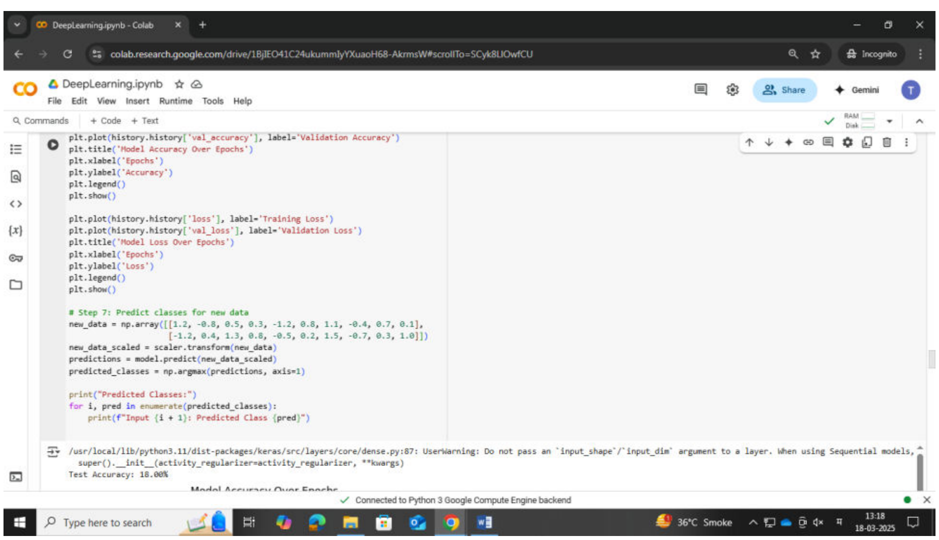
```
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Step 4: Train the model
history = model.fit(X_train, y_train, validation_split=0.2, epochs=50,
batch_size=32, verbose=0)

# Step 5: Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

# Step 6: Visualize training performance
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Step 7: Predict classes for new data
new_data = np.array([[1.2, -0.8, 0.5, 0.3, -1.2, 0.8, 1.1, -0.4, 0.7,
0.1],
                     [-1.2, 0.4, 1.3, 0.8, -0.5, 0.2, 1.5, -0.7, 0.3,
1.0]])
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)
predicted_classes = np.argmax(predictions, axis=1)

print("Predicted Classes:")
for i, pred in enumerate(predicted_classes):
    print(f"Input {i + 1}: Predicted Class {pred}")
```



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

# Step 1: Create a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_classes=3, n_informative=8, random_state=42)
y = y.reshape(-1, 1)  # Reshape labels for encoding

# One-hot encode the labels
# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)  # Use sparse_output instead of sparse
y = encoder.fit_transform(y)

y = encoder.fit_transform(y)

# Split the data 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 2: Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=(X_train.shape[1],)),  # First hidden layer
    tf.keras.layers.Dense(16, activation='relu'),  # Second hidden layer
```



```python
# Step 2: Build the model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(32, activation='relu', input_shape=(X_train.shape[1],)),  # First hidden layer
    tf.keras.layers.Dense(16, activation='relu'),  # Second hidden layer
    tf.keras.layers.Dense(y_train.shape[1], activation='softmax')  # Output layer for multiclass classification
])

# Step 3: Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Step 4: Train the model
history = model.fit(X_train, y_train, validation_split=0.2, epochs=50, batch_size=32, verbose=0)

# Step 5: Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")

# Step 6: Visualize training performance
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss Over Epochs')
plt.xlabel('Epochs')
```

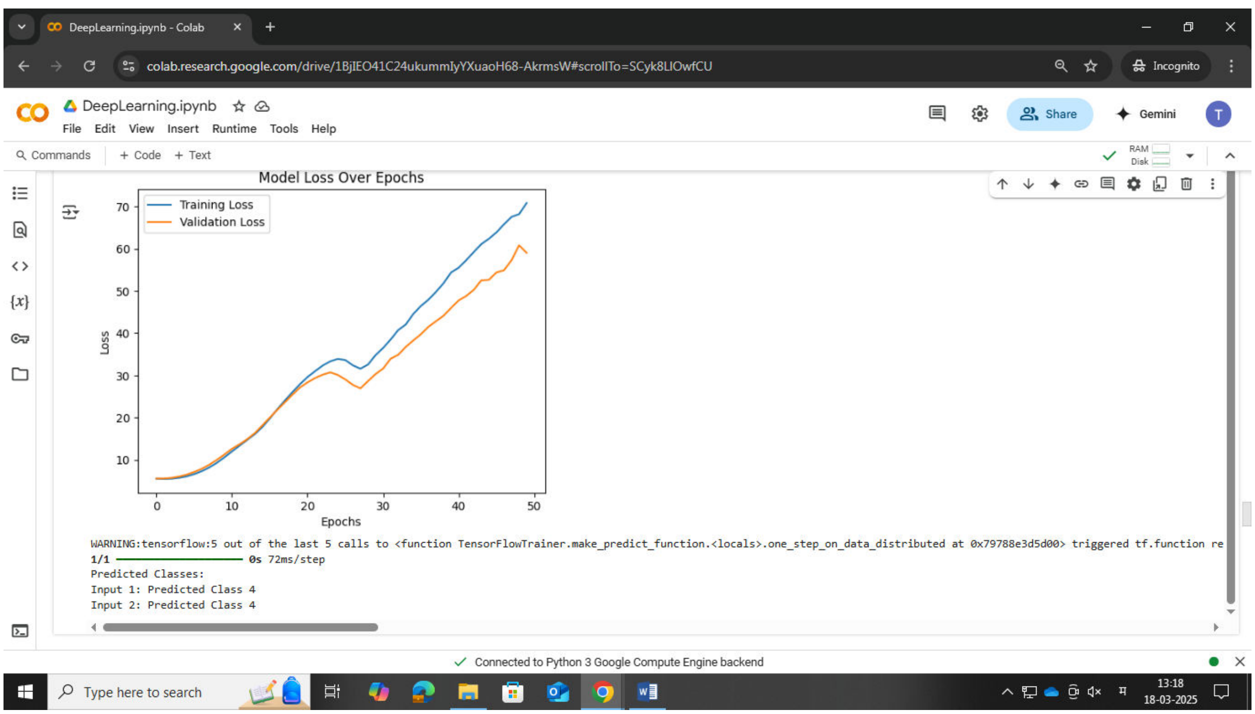


```python
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Step 7: Predict classes for new data
new_data = np.array([[1.2, -0.8, 0.5, 0.3, -1.2, 0.8, 1.1, -0.4, 0.7, 0.1],
                     [-1.2, 0.4, 1.3, 0.8, -0.5, 0.2, 1.5, -0.7, 0.3, 1.0]])
new_data_scaled = scaler.transform(new_data)
predictions = model.predict(new_data_scaled)
predicted_classes = np.argmax(predictions, axis=1)

print("Predicted Classes:")
for i, pred in enumerate(predicted_classes):
    print(f"Input {i + 1}: Predicted Class {pred}")
```

```
/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,
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Test Accuracy: 18.00%
```

Output:


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=SCyk8LIOwfCU
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
/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,
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Test Accuracy: 18.00%
Model Accuracy Over Epochs
Accuracy
Epochs
Training Accuracy
Validation Accuracy
Model Loss Over Epochs
Connected to Python 3 Google Compute Engine backend


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummlyYXuaoH68-AkrmsW#scrollTo=SCyk8LIOwfCU
Model Loss Over Epochs
Training Loss
Validation Loss
Loss
Epochs
WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x79788e3d5d00> triggered tf.function re
1/1 0s 72ms/step
Predicted Classes:
Input 1: Predicted Class 4
Input 2: Predicted Class 4
Connected to Python 3 Google Compute Engine backend

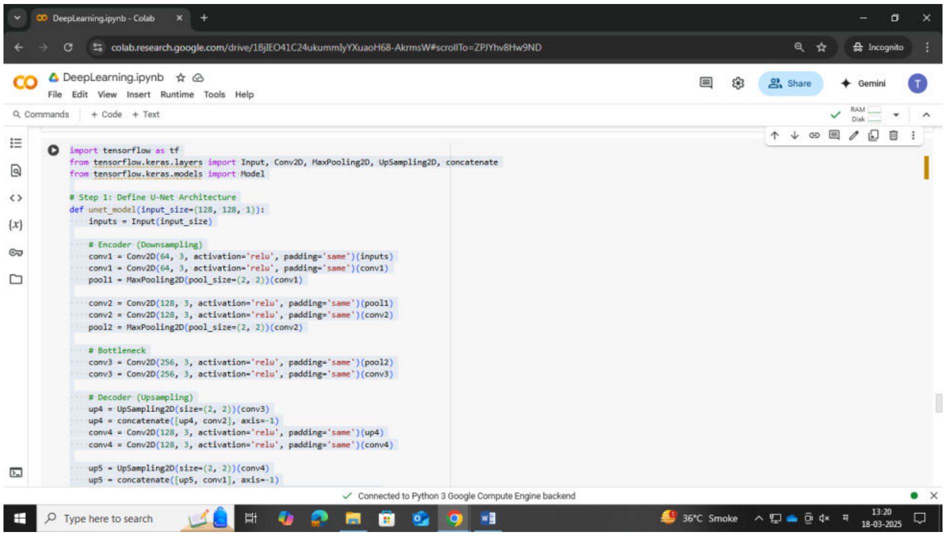
PRACTICAL 4.

Aim: Write a program to implement deep learning Techniques for image segmentation

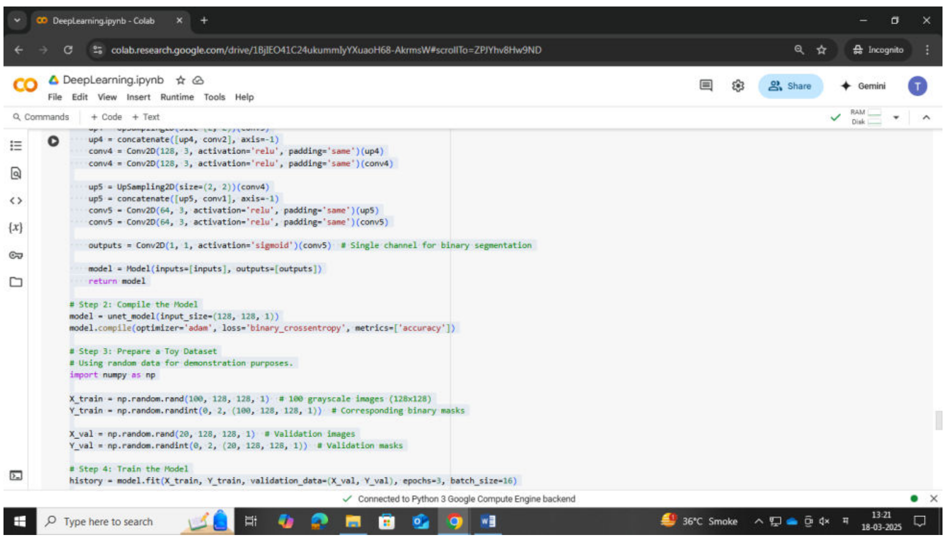
Code:

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D,
UpSampling2D, concatenate
from tensorflow.keras.models import Model

# Step 1: Define U-Net Architecture
def unet_model(input_size=(128, 128, 1)):
    inputs = Input(input_size)

    # Encoder (Downsampling)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv2 = Conv2D(128, 3, activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    # Bottleneck
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(conv3)

    # Decoder (Upsampling)
    up4 = UpSampling2D(size=(2, 2))(conv3)
    up4 = concatenate([up4, conv2], axis=-1)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(up4)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(conv4)

    up5 = UpSampling2D(size=(2, 2))(conv4)
    up5 = concatenate([up5, conv1], axis=-1)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(up5)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(conv5)

    outputs = Conv2D(1, 1, activation='sigmoid')(conv5)  # Single channel
for binary segmentation

    model = Model(inputs=[inputs], outputs=[outputs])
    return model
```

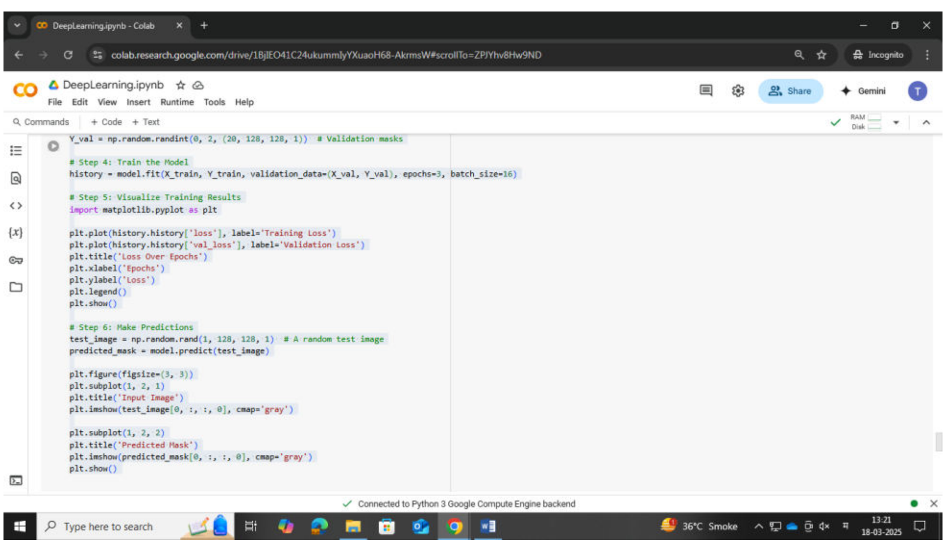
```
# Step 2: Compile the Model
model = unet_model(input_size=(128, 128, 1))
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Step 3: Prepare a Toy Dataset
# Using random data for demonstration purposes.
import numpy as np

X_train = np.random.rand(100, 128, 128, 1)  # 100 grayscale images
(128x128)
Y_train = np.random.randint(0, 2, (100, 128, 128, 1))  # Corresponding
binary masks

X_val = np.random.rand(20, 128, 128, 1)  # Validation images
Y_val = np.random.randint(0, 2, (20, 128, 128, 1))  # Validation masks

# Step 4: Train the Model
history = model.fit(X_train, Y_train, validation_data=(X_val, Y_val),
epochs=3, batch_size=16)

# Step 5: Visualize Training Results
import matplotlib.pyplot as plt
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
# Step 6: Make Predictions
test_image = np.random.rand(1, 128, 128, 1)  # A random test image
predicted_mask = model.predict(test_image)

plt.figure(figsize=(3, 3))
plt.subplot(1, 2, 1)
plt.title('Input Image')
plt.imshow(test_image[0, :, :, 0], cmap='gray')

plt.subplot(1, 2, 2)
plt.title('Predicted Mask')
plt.imshow(predicted_mask[0, :, :, 0], cmap='gray')
plt.show()
```



```python
import tensorflow as tf
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, concatenate
from tensorflow.keras.models import Model

# Step 1: Define U-Net Architecture
def unet_model(input_size=(128, 128, 1)):
    inputs = Input(input_size)

    # Encoder (Downsampling)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(inputs)
    conv1 = Conv2D(64, 3, activation='relu', padding='same')(conv1)
    pool1 = MaxPooling2D(pool_size=(2, 2))(conv1)

    conv2 = Conv2D(128, 3, activation='relu', padding='same')(pool1)
    conv2 = Conv2D(128, 3, activation='relu', padding='same')(conv2)
    pool2 = MaxPooling2D(pool_size=(2, 2))(conv2)

    # Bottleneck
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(pool2)
    conv3 = Conv2D(256, 3, activation='relu', padding='same')(conv3)

    # Decoder (Upsampling)
    up4 = UpSampling2D(size=(2, 2))(conv3)
    up4 = concatenate([up4, conv2], axis=-1)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(up4)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(conv4)

    up5 = UpSampling2D(size=(2, 2))(conv4)
    up5 = concatenate([up5, conv1], axis=-1)
```

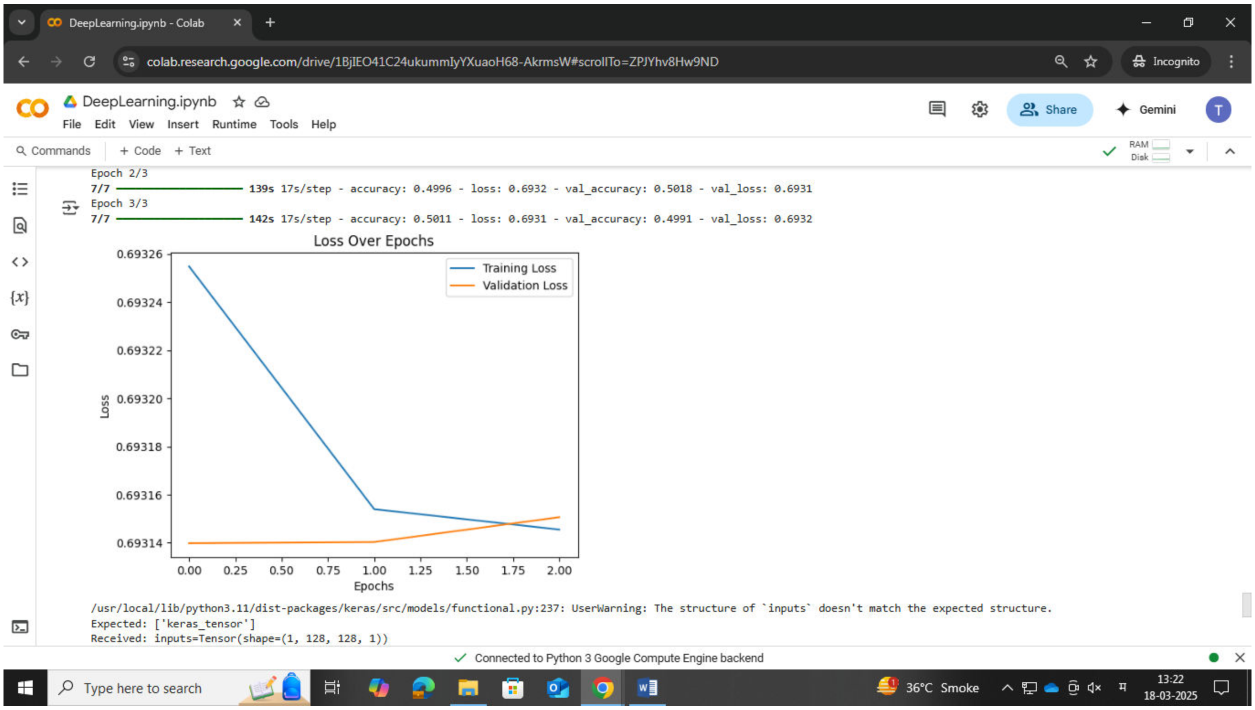


```python
    up4 = concatenate([up4, conv2], axis=-1)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(up4)
    conv4 = Conv2D(128, 3, activation='relu', padding='same')(conv4)

    up5 = UpSampling2D(size=(2, 2))(conv4)
    up5 = concatenate([up5, conv1], axis=-1)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(up5)
    conv5 = Conv2D(64, 3, activation='relu', padding='same')(conv5)

    outputs = Conv2D(1, 1, activation='sigmoid')(conv5)  # Single channel for binary segmentation

    model = Model(inputs=[inputs], outputs=[outputs])
    return model

# Step 2: Compile the Model
model = unet_model(input_size=(128, 128, 1))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Step 3: Prepare a Toy Dataset
# Using random data for demonstration purposes.
import numpy as np

X_train = np.random.rand(100, 128, 128, 1)  # 100 grayscale images (128x128)
Y_train = np.random.randint(0, 2, (100, 128, 128, 1))  # Corresponding binary masks

X_val = np.random.rand(20, 128, 128, 1)  # Validation images
Y_val = np.random.randint(0, 2, (20, 128, 128, 1))  # Validation masks

# Step 4: Train the Model
history = model.fit(X_train, Y_train, validation_data=(X_val, Y_val), epochs=3, batch_size=16)
```



```python
Y_val = np.random.randint(0, 2, (20, 128, 128, 1))  # Validation masks

# Step 4: Train the Model
history = model.fit(X_train, Y_train, validation_data=(X_val, Y_val), epochs=3, batch_size=16)

# Step 5: Visualize Training Results
import matplotlib.pyplot as plt

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss Over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Step 6: Make Predictions
test_image = np.random.rand(1, 128, 128, 1)  # A random test image
predicted_mask = model.predict(test_image)

plt.figure(figsize=(3, 3))
plt.subplot(1, 2, 1)
plt.title('Input Image')
plt.imshow(test_image[0, :, :, 0], cmap='gray')

plt.subplot(1, 2, 2)
plt.title('Predicted Mask')
plt.imshow(predicted_mask[0, :, :, 0], cmap='gray')
plt.show()
```

Output:


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummIyYXuaoH68-AkrmsW#scrollTo=ZPJYhv8Hw9ND
Incognito
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text
Share
Gemini
Epoch 2/3
7/7 139s 17s/step - accuracy: 0.4996 - loss: 0.6932 - val_accuracy: 0.5018 - val_loss: 0.6931
Epoch 3/3
7/7 142s 17s/step - accuracy: 0.5011 - loss: 0.6931 - val_accuracy: 0.4991 - val_loss: 0.6932
Loss Over Epochs
Training Loss
Validation Loss
Loss
Epochs
/usr/local/lib/python3.11/dist-packages/keras/src/models/functional.py:237: UserWarning: The structure of `inputs` doesn't match the expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(1, 128, 128, 1))
Connected to Python 3 Google Compute Engine backend
Type here to search
36°C Smoke
13:22
18-03-2025


DeepLearning.ipynb - Colab
colab.research.google.com/drive/1BjIEO41C24ukummIyYXuaoH68-AkrmsW#scrollTo=ZPJYhv8Hw9ND
Incognito
DeepLearning.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text
Share
Gemini
Epochs
/usr/local/lib/python3.11/dist-packages/keras/src/models/functional.py:237: UserWarning: The structure of `inputs` doesn't match the expected structure.
Expected: ['keras_tensor']
Received: inputs=Tensor(shape=(1, 128, 128, 1))
warnings.warn(msg)
1/1 1s 535ms/step
Input Image
Predicted Mask
Connected to Python 3 Google Compute Engine backend
Type here to search
36°C Smoke
13:22
18-03-2025

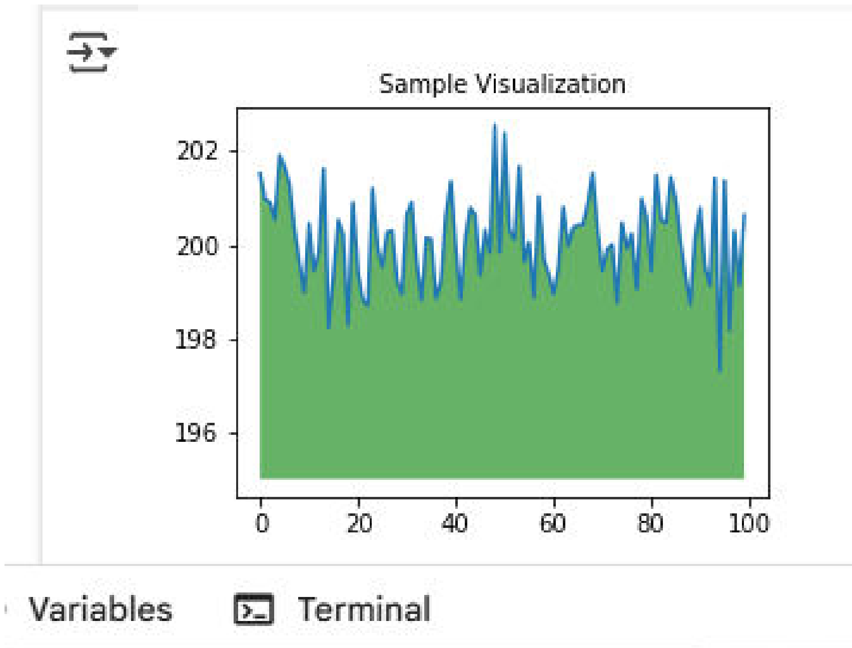
PRACTICAL 5.

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

Code:

```
import numpy as np
import IPython.display as display
from matplotlib import pyplot as plt
import io
import base64

ys = 200 + np.random.randn(100)
x = [x for x in range(len(ys))]

fig = plt.figure(figsize=(4, 3), facecolor='w')
plt.plot(x, ys, '-')
plt.fill_between(x, ys, 195, where=(ys > 195), facecolor='g', alpha=0.6)
plt.title("Sample Visualization", fontsize=10)

data = io.BytesIO()
plt.savefig(data)
image =
F"data:image/png;base64,{base64.b64encode(data.getvalue()).decode()}"
alt = "Sample Visualization"
display.display(display.Markdown(F"""![{alt}]({image})"""))
plt.close(fig)
```

Output:


Sample Visualization
202
200
198
196
0
20
40
60
80
100
Variables
Terminal

PRACTICAL 6.

Aim: Applying the Autoencoder algorithms for encoding real-world data

Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Model
from keras.layers import Input, Dense
from keras.optimizers import Adam
from keras import regularizers

# Step 1: Generate or load the dataset (here, we're using synthetic data)
# Example: 1000 samples, 20 features (could represent customer data,
financial data, etc.)
np.random.seed(42)
data = np.random.rand(1000, 20)

# Step 2: Normalize the data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Step 3: Split data into training and test sets
X_train, X_test = train_test_split(data_scaled, test_size=0.2,
random_state=42)

# Step 4: Define the Autoencoder architecture
input_layer = Input(shape=(X_train.shape[1],))

# Encoder
encoded = Dense(16, activation='relu',
activity_regularizer=regularizers.l2(0.01))(input_layer)
encoded = Dense(8, activation='relu')(encoded)  # Compress the data to 8
features

# Decoder
decoded = Dense(16, activation='relu')(encoded)
decoded = Dense(X_train.shape[1], activation='sigmoid')(decoded)  #
Reconstruct to original dimensions

# Step 5: Build the Autoencoder model
```

```
autoencoder = Model(input_layer, decoded)

# Step 6: Compile the model
autoencoder.compile(optimizer=Adam(), loss='mean_squared_error')

# Step 7: Train the Autoencoder model
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256, shuffle=True,
validation_data=(X_test, X_test))

# Step 8: Get the encoded (compressed) representation of the data
encoder = Model(input_layer, encoded)
encoded_data = encoder.predict(X_test)

# Step 9: Reconstruct the data from the encoded representation
reconstructed_data = autoencoder.predict(X_test)

# Step 10: Calculate the reconstruction error
reconstruction_error = np.mean(np.square(X_test - reconstructed_data),
axis=1)

# Step 11: Detect anomalies based on the reconstruction error
threshold = np.percentile(reconstruction_error, 95)  # Choose an
appropriate threshold
anomalies = reconstruction_error > threshold

# Step 12: Visualize the results
plt.figure(figsize=(10, 6))

# Plot some example data points with reconstruction error
plt.scatter(range(len(reconstruction_error)), reconstruction_error,
c=anomalies, cmap='coolwarm')
plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold')
plt.title('Reconstruction Error with Anomalies Detected')
plt.xlabel('Sample Index')
plt.ylabel('Reconstruction Error')
plt.legend()
plt.show()

# Optionally, you can print out the indices of detected anomalies
print("Anomalies detected at indices:", np.where(anomalies)[0])
```

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from keras.models import Model
from keras.layers import Input, Dense
from keras.optimizers import Adam
from keras import regularizers

# Step 1: Generate or load the dataset (here, we're using synthetic data)
# Example: 1000 samples, 20 features (could represent customer data, financial data, etc.)
np.random.seed(42)
data = np.random.rand(1000, 20)

# Step 2: Normalize the data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data)

# Step 3: Split data into training and test sets
X_train, X_test = train_test_split(data_scaled, test_size=0.2, random_state=42)

# Step 4: Define the Autoencoder architecture
input_layer = Input(shape=(X_train.shape[1],))

# Encoder
encoded = Dense(16, activation='relu', activity_regularizer=regularizers.l2(0.01))(input_layer)
encoded = Dense(8, activation='relu')(encoded)  # Compress the data to 8 features

# Decoder
decoded = Dense(16, activation='relu')(encoded)
decoded = Dense(X_train.shape[1], activation='sigmoid')(decoded)  # Reconstruct to original dimensions
```

Variables Terminal

Commands + Code + Text

```python
# Decoder
decoded = Dense(16, activation='relu')(encoded)
decoded = Dense(X_train.shape[1], activation='sigmoid')(decoded)  # Reconstruct to original dimensions

# Step 5: Build the Autoencoder model
autoencoder = Model(input_layer, decoded)

# Step 6: Compile the model
autoencoder.compile(optimizer=Adam(), loss='mean_squared_error')

# Step 7: Train the Autoencoder model
autoencoder.fit(X_train, X_train, epochs=50, batch_size=256, shuffle=True, validation_data=(X_test, X_test))

# Step 8: Get the encoded (compressed) representation of the data
encoder = Model(input_layer, encoded)
encoded_data = encoder.predict(X_test)

# Step 9: Reconstruct the data from the encoded representation
reconstructed_data = autoencoder.predict(X_test)

# Step 10: Calculate the reconstruction error
reconstruction_error = np.mean(np.square(X_test - reconstructed_data), axis=1)

# Step 11: Detect anomalies based on the reconstruction error
threshold = np.percentile(reconstruction_error, 95)  # Choose an appropriate threshold
anomalies = reconstruction_error > threshold

# Step 12: Visualize the results
plt.figure(figsize=(10, 6))

# Plot some example data points with reconstruction error
plt.scatter(range(len(reconstruction_error)), reconstruction_error, c=anomalies, cmap='coolwarm')
```

Variables Terminal

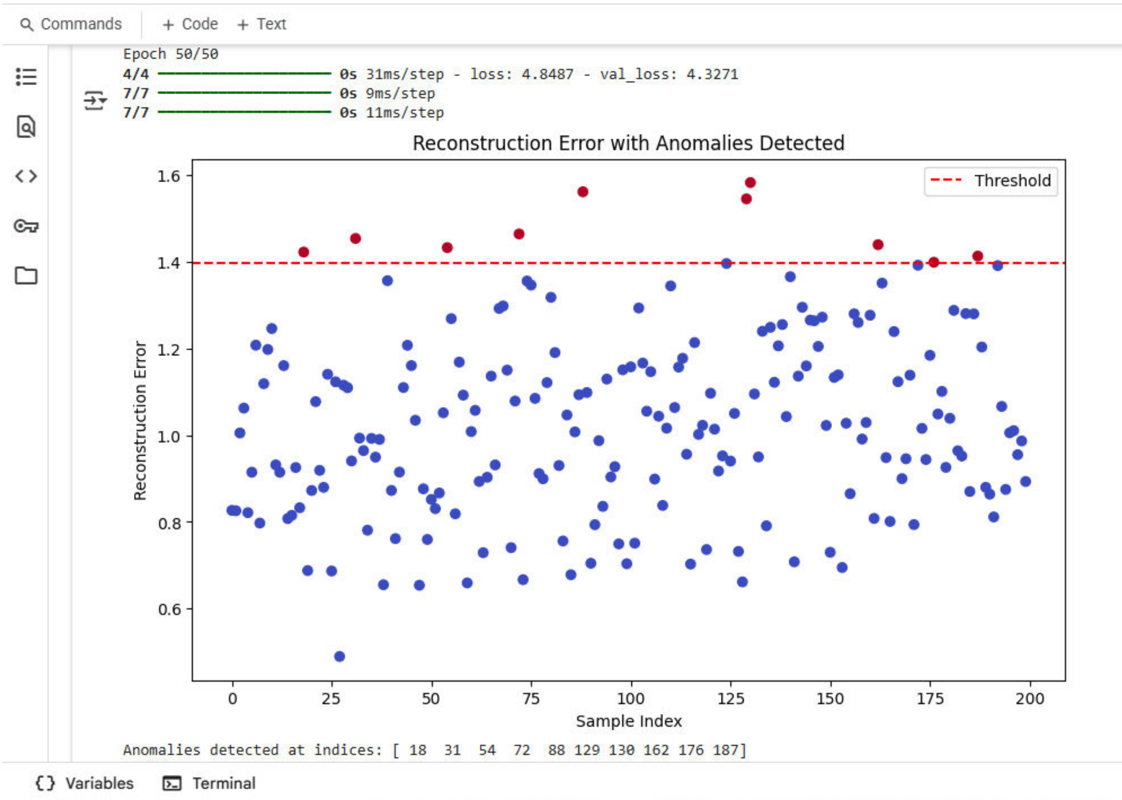
```python
anomalies = reconstruction_error > threshold

# Step 12: Visualize the results
plt.figure(figsize=(10, 6))

# Plot some example data points with reconstruction error
plt.scatter(range(len(reconstruction_error)), reconstruction_error, c=anomalies, cmap='coolwarm')
plt.axhline(y=threshold, color='r', linestyle='--', label='Threshold')
plt.title('Reconstruction Error with Anomalies Detected')
plt.xlabel('Sample Index')
plt.ylabel('Reconstruction Error')
plt.legend()
plt.show()

# Optionally, you can print out the indices of detected anomalies
print("Anomalies detected at indices:", np.where(anomalies)[0])
```

```
Epoch 1/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 3s 105ms/step - loss: 22.5314 - val_loss: 18.9008
Epoch 2/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 21.8574 - val_loss: 18.3640
Epoch 3/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 21.5065 - val_loss: 17.8438
Epoch 4/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 20.6853 - val_loss: 17.3378
Epoch 5/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 20.1411 - val_loss: 16.8436
Epoch 6/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 19.2404 - val_loss: 16.3619
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 19.3938 - val_loss: 15.8894
Epoch 8/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 18.2369 - val_loss: 15.4306
Epoch 9/50
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 19.2404 - val_loss: 16.3619
Epoch 7/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 19.3938 - val_loss: 15.8894
Epoch 8/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 18.2369 - val_loss: 15.4306
Epoch 9/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 17.7372 - val_loss: 14.9819
Epoch 10/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 17.4574 - val_loss: 14.5441
Epoch 11/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 38ms/step - loss: 16.7150 - val_loss: 14.1189
Epoch 12/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - loss: 16.0792 - val_loss: 13.7036
Epoch 13/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 15.6877 - val_loss: 13.3001
Epoch 14/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 15.3827 - val_loss: 12.9067
Epoch 15/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 14.7938 - val_loss: 12.5243
Epoch 16/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 14.0438 - val_loss: 12.1523
Epoch 17/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 13.8878 - val_loss: 11.7897
Epoch 18/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 13.4725 - val_loss: 11.4371
Epoch 19/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 13.1336 - val_loss: 11.0936
Epoch 20/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 12.4126 - val_loss: 10.7611
Epoch 21/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 12.2853 - val_loss: 10.4371
Epoch 22/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 11.9133 - val_loss: 10.1208
Epoch 23/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - loss: 11.5012 - val_loss: 9.8146
```

```
Epoch 24/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - loss: 11.1584 - val_loss: 9.5163
Epoch 25/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 10.6300 - val_loss: 9.2265
Epoch 26/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 10.3495 - val_loss: 8.9447
Epoch 27/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 9.8748 - val_loss: 8.6720
Epoch 28/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 9.6015 - val_loss: 8.4068
Epoch 29/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 9.5022 - val_loss: 8.1491
Epoch 30/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 9.0660 - val_loss: 7.8998
Epoch 31/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 8.7605 - val_loss: 7.6586
Epoch 32/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 8.5331 - val_loss: 7.4241
Epoch 33/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 8.3703 - val_loss: 7.1962
Epoch 34/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 7.9267 - val_loss: 6.9774
Epoch 35/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 7.7720 - val_loss: 6.7651
Epoch 36/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 7.5597 - val_loss: 6.5600
Epoch 37/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - loss: 7.1776 - val_loss: 6.3624
Epoch 38/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 27ms/step - loss: 7.1675 - val_loss: 6.1703
Epoch 39/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 6.8353 - val_loss: 5.9860
Epoch 40/50
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 23ms/step - loss: 6.5669 - val_loss: 5.8074
Epoch 41/50
```

Output:


Commands
+ Code + Text
Epoch 50/50
4/4 0s 31ms/step - loss: 4.8487 - val_loss: 4.3271
7/7 0s 9ms/step
7/7 0s 11ms/step
Reconstruction Error with Anomalies Detected
Threshold
Reconstruction Error
Sample Index
1.6
1.4
1.2
1.0
0.8
0.6
0
25
50
75
100
125
150
175
200
Anomalies detected at indices: [18 31 54 72 88 129 130 162 176 187]
Variables
Terminal

PRACTICAL 7.

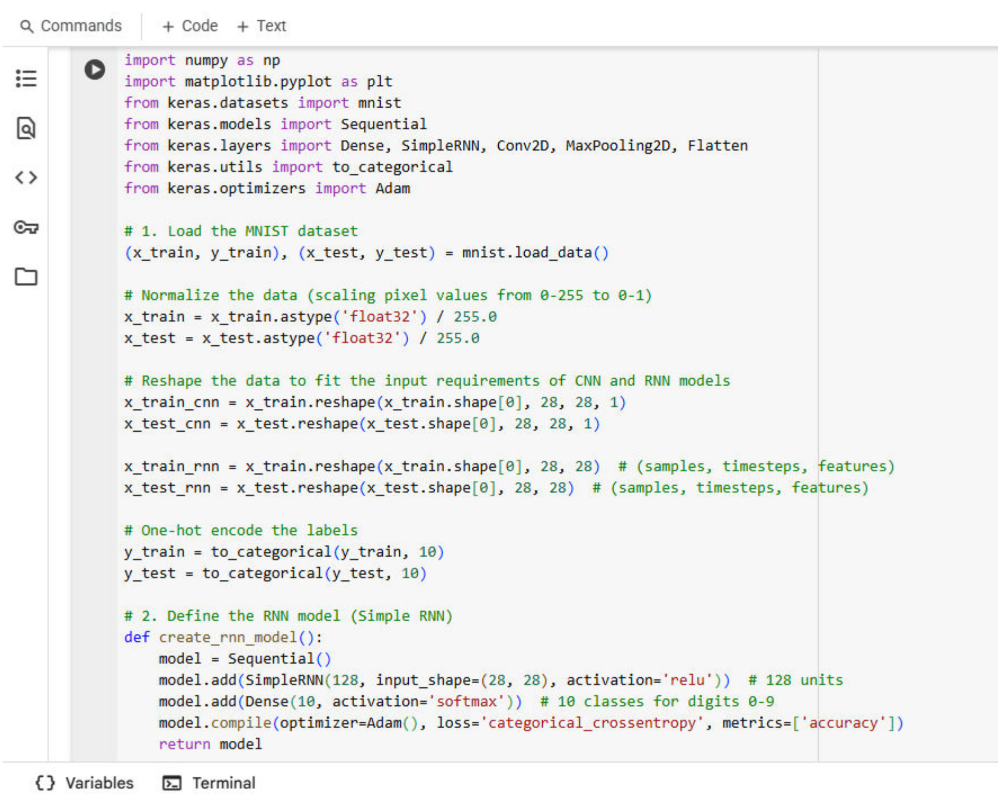
Aim: Write a program for character recognition using RNN and compare it with CNN.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, Conv2D, MaxPooling2D, Flatten
from keras.utils import to_categorical
from keras.optimizers import Adam

# 1. Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the data (scaling pixel values from 0-255 to 0-1)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Reshape the data to fit the input requirements of CNN and RNN models
x_train_cnn = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test_cnn = x_test.reshape(x_test.shape[0], 28, 28, 1)

x_train_rnn = x_train.reshape(x_train.shape[0], 28, 28)  # (samples, 
timesteps, features)
x_test_rnn = x_test.reshape(x_test.shape[0], 28, 28)  # (samples, 
timesteps, features)

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 2. Define the RNN model (Simple RNN)
def create_rnn_model():
    model = Sequential()
    model.add(SimpleRNN(128, input_shape=(28, 28), activation='relu'))  # 
128 units
    model.add(Dense(10, activation='softmax'))  # 10 classes for digits 0-
9
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', 
metrics=['accuracy'])
    return model
```

```
# 3. Define the CNN model
def create_cnn_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu',
input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(10, activation='softmax'))  # 10 classes for digits 0-
9
    model.compile(optimizer=Adam(), loss='categorical_crossentropy',
metrics=['accuracy'])
    return model

# 4. Train the RNN model
rnn_model = create_rnn_model()
rnn_history = rnn_model.fit(x_train_rnn, y_train, epochs=5,
batch_size=128, validation_data=(x_test_rnn, y_test))

# 5. Train the CNN model
cnn_model = create_cnn_model()
cnn_history = cnn_model.fit(x_train_cnn, y_train, epochs=5,
batch_size=128, validation_data=(x_test_cnn, y_test))

# 6. Evaluate the models
rnn_test_loss, rnn_test_acc = rnn_model.evaluate(x_test_rnn, y_test,
verbose=0)
cnn_test_loss, cnn_test_acc = cnn_model.evaluate(x_test_cnn, y_test,
verbose=0)

# Print the results
print(f"RNN Model - Test Accuracy: {rnn_test_acc*100:.2f}%")
print(f"CNN Model - Test Accuracy: {cnn_test_acc*100:.2f}%")

# 7. Plot Training and Validation Accuracy for Both Models
plt.figure(figsize=(12, 6))

# RNN Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(rnn_history.history['accuracy'], label='Training Accuracy')
plt.plot(rnn_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('RNN Model Accuracy')
```

```
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# CNN Accuracy Plot
plt.subplot(1, 2, 2)
plt.plot(cnn_history.history['accuracy'], label='Training Accuracy')
plt.plot(cnn_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('CNN Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```




```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, Conv2D, MaxPooling2D, Flatten
from keras.utils import to_categorical
from keras.optimizers import Adam

# 1. Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize the data (scaling pixel values from 0-255 to 0-1)
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Reshape the data to fit the input requirements of CNN and RNN models
x_train_cnn = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test_cnn = x_test.reshape(x_test.shape[0], 28, 28, 1)

x_train_rnn = x_train.reshape(x_train.shape[0], 28, 28)  # (samples, timesteps, features)
x_test_rnn = x_test.reshape(x_test.shape[0], 28, 28)  # (samples, timesteps, features)

# One-hot encode the labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 2. Define the RNN model (Simple RNN)
def create_rnn_model():
    model = Sequential()
    model.add(SimpleRNN(128, input_shape=(28, 28), activation='relu'))  # 128 units
    model.add(Dense(10, activation='softmax'))  # 10 classes for digits 0-9
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Commands + Code + Text

```python
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# 3. Define the CNN model
def create_cnn_model():
    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Conv2D(64, kernel_size=(3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(10, activation='softmax'))  # 10 classes for digits 0-9
    model.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# 4. Train the RNN model
rnn_model = create_rnn_model()
rnn_history = rnn_model.fit(x_train_rnn, y_train, epochs=5, batch_size=128, validation_data=(x_test_rnn, y_test))

# 5. Train the CNN model
cnn_model = create_cnn_model()
cnn_history = cnn_model.fit(x_train_cnn, y_train, epochs=5, batch_size=128, validation_data=(x_test_cnn, y_test))

# 6. Evaluate the models
rnn_test_loss, rnn_test_acc = rnn_model.evaluate(x_test_rnn, y_test, verbose=0)
cnn_test_loss, cnn_test_acc = cnn_model.evaluate(x_test_cnn, y_test, verbose=0)

# Print the results
print(f"RNN Model - Test Accuracy: {rnn_test_acc*100:.2f}%")
print(f"CNN Model - Test Accuracy: {cnn_test_acc*100:.2f}%")

# 7. Plot Training and Validation Accuracy for Both Models
plt.figure(figsize=(12, 6))
```

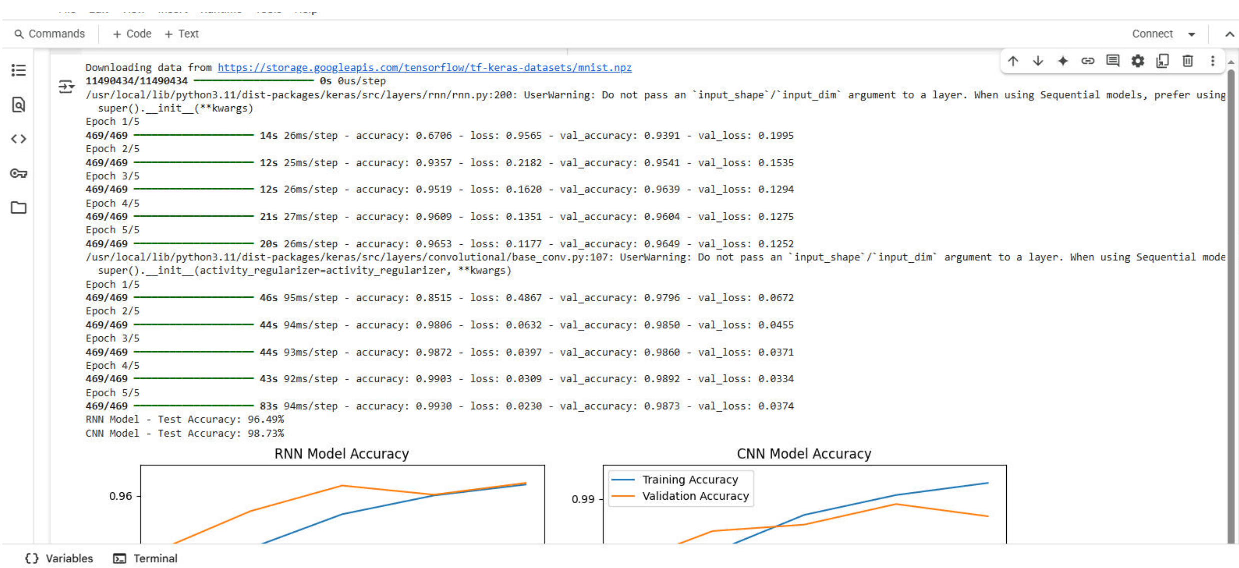
{} Variables Terminal

Commands + Code + Text Connect

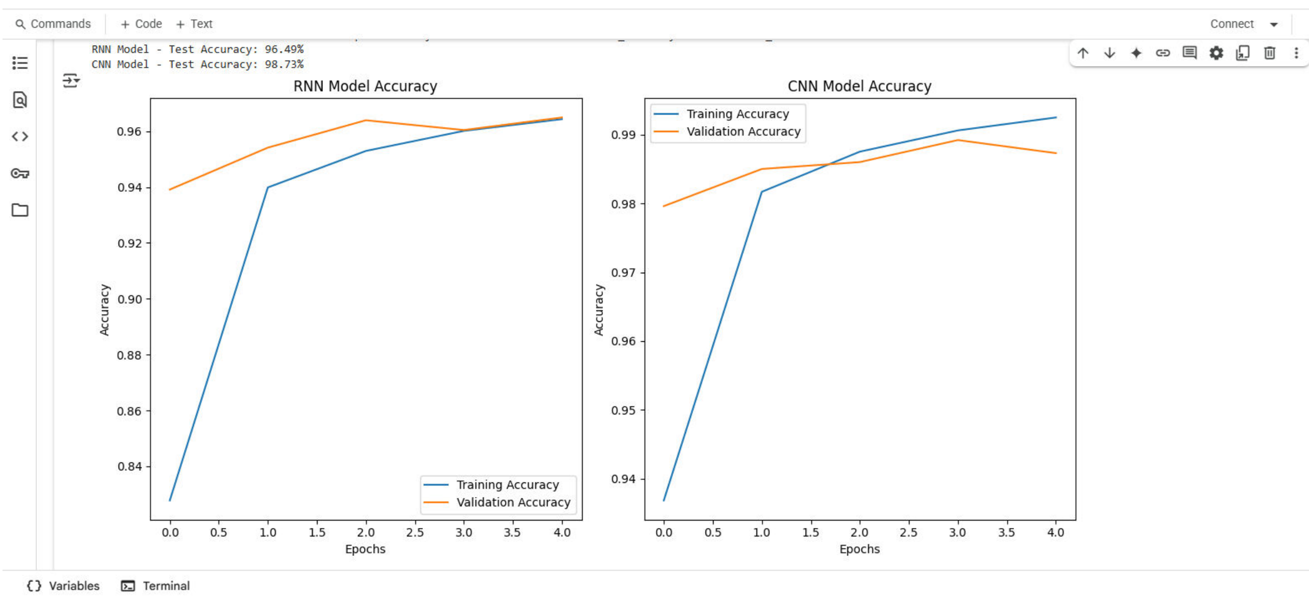
```python
# 7. Plot Training and Validation Accuracy for Both Models
plt.figure(figsize=(12, 6))

# RNN Accuracy Plot
plt.subplot(1, 2, 1)
plt.plot(rnn_history.history['accuracy'], label='Training Accuracy')
plt.plot(rnn_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('RNN Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

# CNN Accuracy Plot
plt.subplot(1, 2, 2)
plt.plot(cnn_history.history['accuracy'], label='Training Accuracy')
plt.plot(cnn_history.history['val_accuracy'], label='Validation Accuracy')
plt.title('CNN Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```

```
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/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using
  super().__init__(**kwargs)
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 14s 26ms/step - accuracy: 0.6706 - loss: 0.9565 - val_accuracy: 0.9391 - val_loss: 0.1995
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9357 - loss: 0.2182 - val_accuracy: 0.9541 - val_loss: 0.1535
Epoch 3/5
```

{} Variables Terminal


Commands
+ Code
+ Text
Connect
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/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using
super().__init__(**kwargs)
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 14s 26ms/step - accuracy: 0.6706 - loss: 0.9565 - val_accuracy: 0.9391 - val_loss: 0.1995
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 25ms/step - accuracy: 0.9357 - loss: 0.2182 - val_accuracy: 0.9541 - val_loss: 0.1535
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 12s 26ms/step - accuracy: 0.9519 - loss: 0.1620 - val_accuracy: 0.9639 - val_loss: 0.1294
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 21s 27ms/step - accuracy: 0.9609 - loss: 0.1351 - val_accuracy: 0.9604 - val_loss: 0.1275
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 20s 26ms/step - accuracy: 0.9653 - loss: 0.1177 - val_accuracy: 0.9649 - val_loss: 0.1252
/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 mode
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 46s 95ms/step - accuracy: 0.8515 - loss: 0.4867 - val_accuracy: 0.9796 - val_loss: 0.0672
Epoch 2/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 44s 94ms/step - accuracy: 0.9806 - loss: 0.0632 - val_accuracy: 0.9850 - val_loss: 0.0455
Epoch 3/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 44s 93ms/step - accuracy: 0.9872 - loss: 0.0397 - val_accuracy: 0.9860 - val_loss: 0.0371
Epoch 4/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 43s 92ms/step - accuracy: 0.9903 - loss: 0.0309 - val_accuracy: 0.9892 - val_loss: 0.0334
Epoch 5/5
469/469 ━━━━━━━━━━━━━━━━━━━━ 83s 94ms/step - accuracy: 0.9930 - loss: 0.0230 - val_accuracy: 0.9873 - val_loss: 0.0374
RNN Model - Test Accuracy: 96.49%
CNN Model - Test Accuracy: 98.73%
RNN Model Accuracy
0.96
CNN Model Accuracy
Training Accuracy
Validation Accuracy
0.99
Variables
Terminal

Output:


Commands
+ Code
+ Text
Connect
RNN Model - Test Accuracy: 96.49%
CNN Model - Test Accuracy: 98.73%
RNN Model Accuracy
Accuracy
0.96
0.94
0.92
0.90
0.88
0.86
0.84
Training Accuracy
Validation Accuracy
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epochs
CNN Model Accuracy
Training Accuracy
Validation Accuracy
Accuracy
0.99
0.98
0.97
0.96
0.95
0.94
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Epochs
Variables
Terminal

PRACTICAL 8.

Aim: Write a program to develop Autoencoders using MNIST Handwritten Digits.

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense
from keras.optimizers import Adam
from keras.utils import plot_model

# Step 1: Load and preprocess the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize the images to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images to 1D vectors (28x28 -> 784)
x_train = x_train.reshape((x_train.shape[0], 784))
x_test = x_test.reshape((x_test.shape[0], 784))

# Step 2: Build the Autoencoder model
input_img = Input(shape=(784,))

# Encoder: Compress the data to a lower-dimensional space
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)  # Bottleneck layer
(compressed representation)

# Decoder: Reconstruct the data back to its original shape
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)  # Output should have
the same shape as input (784)

# Combine the encoder and decoder to form the autoencoder
autoencoder = Model(input_img, decoded)

# Step 3: Compile the model
autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')
```

```
# Step 4: Train the Autoencoder
autoencoder.fit(x_train, x_train, epochs=20, batch_size=256, shuffle=True,
validation_data=(x_test, x_test))

# Step 5: Evaluate the Autoencoder's performance on the test set
decoded_imgs = autoencoder.predict(x_test)

# Step 6: Visualize the original and reconstructed images
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    ax.set_title("Original")
    ax.axis('off')

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    ax.set_title("Reconstructed")
    ax.axis('off')

plt.show()

# Optionally, save the model architecture visualization
# plot_model(autoencoder, to_file='autoencoder_model.png',
show_shapes=True)
```

```
import numpy as np
import matplotlib.pyplot as plt
from keras.datasets import mnist
from keras.models import Model
from keras.layers import Input, Dense
from keras.optimizers import Adam
from keras.utils import plot_model

# Step 1: Load and preprocess the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

# Normalize the images to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Flatten the images to 1D vectors (28x28 -> 784)
x_train = x_train.reshape((x_train.shape[0], 784))
x_test = x_test.reshape((x_test.shape[0], 784))

# Step 2: Build the Autoencoder model
input_img = Input(shape=(784,))

# Encoder: Compress the data to a lower-dimensional space
encoded = Dense(128, activation='relu')(input_img)
encoded = Dense(64, activation='relu')(encoded)
encoded = Dense(32, activation='relu')(encoded)  # Bottleneck layer (compressed representation)

# Decoder: Reconstruct the data back to its original shape
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
```

Commands + Code + Text

```
# Decoder: Reconstruct the data back to its original shape
decoded = Dense(64, activation='relu')(encoded)
decoded = Dense(128, activation='relu')(decoded)
decoded = Dense(784, activation='sigmoid')(decoded)  # Output should have the same shape as input (784)

# Combine the encoder and decoder to form the autoencoder
autoencoder = Model(input_img, decoded)

# Step 3: Compile the model
autoencoder.compile(optimizer=Adam(), loss='binary_crossentropy')

# Step 4: Train the Autoencoder
autoencoder.fit(x_train, x_train, epochs=20, batch_size=256, shuffle=True, validation_data=(x_test, x_test))

# Step 5: Evaluate the Autoencoder's performance on the test set
decoded_imgs = autoencoder.predict(x_test)

# Step 6: Visualize the original and reconstructed images
n = 10  # How many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Display original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    ax.set_title("Original")
    ax.axis('off')

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    ax.set_title("Reconstructed")
    ax.axis('off')
```

Variables Terminal

Commands + Code + Text

```
    ax.axis('off')

    # Display reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(28, 28), cmap='gray')
    ax.set_title("Reconstructed")
    ax.axis('off')

plt.show()

# Optionally, save the model architecture visualization
# plot_model(autoencoder, to_file='autoencoder_model.png', show_shapes=True)
```

```
Epoch 1/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 8s 20ms/step - loss: 0.3511 - val_loss: 0.1742
Epoch 2/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.1650 - val_loss: 0.1403
Epoch 3/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 22ms/step - loss: 0.1376 - val_loss: 0.1281
Epoch 4/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 9s 15ms/step - loss: 0.1270 - val_loss: 0.1215
Epoch 5/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 20ms/step - loss: 0.1204 - val_loss: 0.1152
Epoch 6/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.1158 - val_loss: 0.1111
Epoch 7/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.1114 - val_loss: 0.1073
Epoch 8/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - loss: 0.1083 - val_loss: 0.1051
Epoch 9/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.1060 - val_loss: 0.1046
Epoch 10/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.1046 - val_loss: 0.1025
Epoch 11/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 18ms/step - loss: 0.1030 - val_loss: 0.1011
```

Variables Terminal

Output:


Commands
+ Code
+ Text
Connect
Epoch 13/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 19ms/step - loss: 0.1006 - val_loss: 0.0988
Epoch 14/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.0994 - val_loss: 0.0975
Epoch 15/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step - loss: 0.0981 - val_loss: 0.0961
Epoch 16/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 19ms/step - loss: 0.0970 - val_loss: 0.0949
Epoch 17/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.0960 - val_loss: 0.0945
Epoch 18/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 17ms/step - loss: 0.0948 - val_loss: 0.0936
Epoch 19/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0945 - val_loss: 0.0932
Epoch 20/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - loss: 0.0942 - val_loss: 0.0929
WARNING:tensorflow:5 out of the last 15 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x7b8da1473e20> triggered tf.function retracing. T
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Original
Reconstructed
Variables
Terminal

PRACTICAL 9.

Aim: 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 yfinance as yf
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense

# Step 1: Load historical stock data
data = yf.download('GOOG', start='2010-01-01', end='2023-12-31')
stock_prices = data['Close'].values.reshape(-1, 1)  # Use 'Close' price

# Step 2: Normalize prices
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_prices = scaler.fit_transform(stock_prices)

# Step 3: Create sequences for training (60 days to predict next day)
X = []
y = []
sequence_length = 60

for i in range(sequence_length, len(scaled_prices)):
    X.append(scaled_prices[i-sequence_length:i])
    y.append(scaled_prices[i])

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

# Step 4: Split into training and testing sets (80% training)
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# Step 5: Build the LSTM model
model = Sequential([
    LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],
1)),
    LSTM(units=50),
```

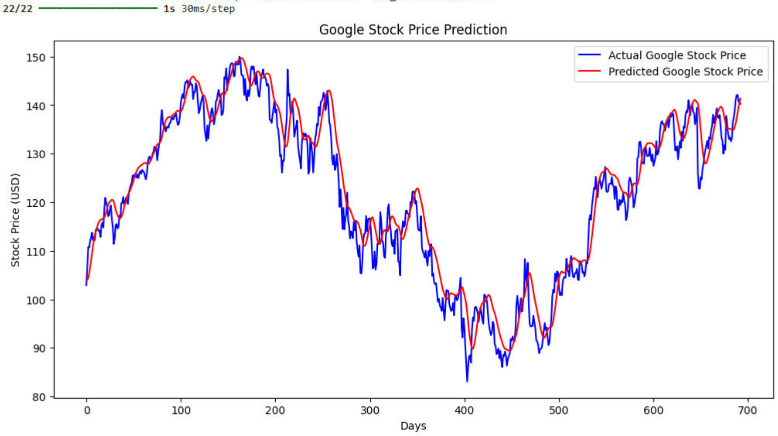
```
    Dense(units=1)
])

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

# Step 6: Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_data=(X_test, y_test))

# Step 7: Predict and inverse scale
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)
actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Step 8: Plot results
plt.figure(figsize=(12, 6))
plt.plot(actual, color='blue', label='Actual Google Stock Price')
plt.plot(predictions, color='red', label='Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel('Days')
plt.ylabel('Stock Price (USD)')
plt.legend()
plt.show()
```

```python
# Step 4: Split into training and testing sets (80% training)
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# Step 5: Build the LSTM model
model = Sequential([
    LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1], 1)),
    LSTM(units=50),
    Dense(units=1)
])

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

# Step 6: Train the model
model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))

# Step 7: Predict and inverse scale
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)
actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Step 8: Plot results
plt.figure(figsize=(12, 6))
plt.plot(actual, color='blue', label='Actual Google Stock Price')
plt.plot(predictions, color='red', label='Predicted Google Stock Price')
plt.title('Google Stock Price Prediction')
plt.xlabel('Days')
plt.ylabel('Stock Price (USD)')
plt.legend()
plt.show()
```

```
[*********************100%***********************]  1 of 1 completedEpoch 1/10
/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 a
  super().__init__(**kwargs)
87/87 ━━━━━━━━━━━━━━━━━━━━ 8s 55ms/step - loss: 0.0090 - val_loss: 0.0013
Epoch 2/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 6s 66ms/step - loss: 1.5237e-04 - val_loss: 0.0012
Epoch 3/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 10s 65ms/step - loss: 1.2637e-04 - val_loss: 0.0011
Epoch 4/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 11s 69ms/step - loss: 1.2209e-04 - val_loss: 0.0012
Epoch 5/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 5s 57ms/step - loss: 1.1353e-04 - val_loss: 0.0012
Epoch 6/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 5s 50ms/step - loss: 1.3336e-04 - val_loss: 0.0010
Epoch 7/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 5s 63ms/step - loss: 1.1695e-04 - val_loss: 9.6799e-04
Epoch 8/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 9s 51ms/step - loss: 1.3559e-04 - val_loss: 0.0011
Epoch 9/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 5s 61ms/step - loss: 1.1140e-04 - val_loss: 8.5251e-04
Epoch 10/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 11s 66ms/step - loss: 1.1321e-04 - val_loss: 9.2862e-04
22/22 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step
```

Output:

```
Epoch 10/10
87/87 ━━━━━━━━━━━━━━━━━━━━ 11s 66ms/step - loss: 1.1321e-04 - val_loss: 9.2862e-04
22/22 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step
```



PRACTICAL 10.

Aim: Applying Generative Adversarial Networks for image generation and unsupervised tasks.

Code:

STEP 1: Import Libraries

```
import tensorflow as tf
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt
import os
```

STEP 2: Load CIFAR-10 Dataset

```
(x_train, _), (_, _) = tf.keras.datasets.cifar10.load_data()
x_train = x_train.astype('float32')
x_train = (x_train - 127.5) / 127.5  # Normalize to [-1, 1]
BUFFER_SIZE = 50000
BATCH_SIZE = 128
train_dataset =
tf.data.Dataset.from_tensor_slices(x_train).shuffle(BUFFER_SIZE).batch(BATCH_SIZE)
```

STEP 3: Define the Generator

```
def make_generator_model():
    model = tf.keras.Sequential([
        layers.Dense(8*8*256, use_bias=False, input_shape=(100,)),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Reshape((8, 8, 256)),
        layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False),
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False),
```

```
        layers.BatchNormalization(),
        layers.LeakyReLU(),
        layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', use_bias=False,
activation='tanh')
    ])
    return model
generator = make_generator_model()
```

STEP 4: Define the Discriminator

```
def make_discriminator_model():
    model = tf.keras.Sequential([
        layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same', input_shape=[32, 32, 3]),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'),
        layers.LeakyReLU(),
        layers.Dropout(0.3),

        layers.Flatten(),
        layers.Dense(1)
    ])
    return model

discriminator = make_discriminator_model()
```

STEP 5: Loss Functions & Optimizers

```
cross_entropy = tf.keras.losses.BinaryCrossentropy(from_logits=True)
def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
def discriminator_loss(real_output, fake_output):
```

```
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    return real_loss + fake_loss
generator_optimizer = tf.keras.optimizers.Adam(1e-4)
discriminator_optimizer = tf.keras.optimizers.Adam(1e-4)
```

STEP 6: Define Training Loop

```
EPOCHS = 50
noise_dim = 100
num_examples_to_generate = 16
seed = tf.random.normal([num_examples_to_generate, noise_dim])
@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise, training=True)
        real_output = discriminator(images, training=True)
        fake_output = discriminator(generated_images, training=True)
        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output, fake_output)
    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)
    generator_optimizer.apply_gradients(zip(gradients_of_generator,
generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator,
discriminator.trainable_variables))
def generate_and_save_images(model, epoch, test_input):
    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)
        img = (predictions[i] + 1.0) / 2.0  # Rescale from [-1,1] to [0,1]
        plt.imshow(img)
        plt.axis('off')
    plt.suptitle(f'Epoch {epoch}', fontsize=16)
    plt.show()
def train(dataset, epochs):
    for epoch in range(1, epochs + 1):
        for image_batch in dataset:
            train_step(image_batch)
        if epoch % 5 == 0 or epoch == 1:
            print(f' Epoch {epoch} completed!')
            generate_and_save_images(generator, epoch, seed)
```

STEP 7: Start Training

```
train(train_dataset, EPOCHS)
```