P.T.LEE CHENGALVARAYA NAICKER COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai)

Vallal P.T.Lee Chengalvaraya Naicker Nagar, Oovery, Kanchipuram – 631 502

Department of Computer Science and Engineering

Regulation - 2021

CCS355 –Neural Netwok and Deep Learning

RECORD

NAME :

REG.NO :

YEAR/SEMESTER :

P.T.LEE CHENGALVARAYA NAICKER COLLEGE OF ENGINEERING & TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai)

Vallal P.T.Lee Chengalvaraya Naicker Nagar, Oovery, Kanchipuram – 631 502

BONAFIDE CERTIFICATE

This is to certify that record work done by_____with Reg. No: _____of_____Year_____Semester Bachelor of Computer Science and Engineering, in the **CCS355-NEURAL NETWORK AND DEEP LEARNING LABORATORY** during the academic year2023-24.

Lab-in-charge Head of the Department

Submitted for the Practical Examination held on _____

Internal Examiner External Examiner

SI.NO	DATE	LAB EXPERIMENTS	PAGE NO	MARK	SIGN
1		Implement simple vector addition in TensorFlow			
2		Implement a regression model in Keras			
3		Implement a perceptron in TensorFlow/Keras Environment			
4		Implement a Feed-Forward Network in TensorFlow/Keras			
5		Implement an Image Classifier using CNN in TensorFlow/Keras			
6		Improve the Deep learning model by fine tuning hyper parameters			
7		Implement a Transfer Learning concept in Image Classification			
8		Using a pre trained model on Keras for Transfer Learning			
9		Perform Sentiment Analysis using RNN			
10		Implement an LSTM based Autoencoder in TensorFlow/Keras			
11		Image generation using GAN			

SI.NO	DATE	Additional Experiments	PAGE NO	MARK	SIGN
12		Train a Deep learning model to classify a given image using pre trained model			
13		Recommendation system from sales data using Deep Learning			
14		Implement Object Detection using CNN			
15		Implement any simple Reinforcement Algorithm for an NLP problem			

LAB EXPERIMENTS:

1. Implement simple vector addition in TensorFlow.
2. Implement a regression model in Keras.
3. Implement a perceptron in TensorFlow/Keras Environment.
4. Implement a Feed-Forward Network in TensorFlow/Keras.
5. Implement an Image Classifier using CNN in TensorFlow/Keras.
6. Improve the Deep learning model by fine tuning hyper parameters.
7. Implement a Transfer Learning concept in Image Classification.
8. Using a pre trained model on Keras for Transfer Learning
9. Perform Sentiment Analysis using RNN
10. Implement an LSTM based Autoencoder in TensorFlow/Keras.
11. Image generation using GAN

Additional Experiments:

12. Train a Deep learning model to classify a given image using pre trained model
13. Recommendation system from sales data using Deep Learning
14. Implement Object Detection using CNN
15. Implement any simple Reinforcement Algorithm for an NLP problem

INSTALLATION & PROCEDURE

The execution of the experiments mentioned can be done using a variety of environments, and the choice depends on your preferences and requirements. Here's a breakdown:

1. **Jupyter Notebooks:**
    - Jupyter Notebooks are interactive documents that allow you to write and execute code in a step-by-step manner.
    - They are widely used for data analysis, machine learning, and experimentation.
    - You can install Jupyter using Anaconda or pip.
2. **Anaconda:**
    - Anaconda is a distribution of Python that comes with pre-installed scientific packages and tools.
    - It includes popular libraries like NumPy, pandas, scikit-learn, and more.

- While Anaconda itself is not necessary, it can simplify the setup of your Python environment.

3. **Google Colab:**
    - Google Colab is a cloud-based Jupyter notebook environment provided by Google.
    - It allows you to run code in the cloud and provides access to GPU/TPU for free.
    - Colab is suitable for resource-intensive tasks like deep learning.
4. **Installation of Libraries:**
    - Most of the experiments listed involve TensorFlow and Keras, which can be installed using **pip** in any Python environment.
    - Virtual environments can be used to manage dependencies for each experiment.
5. **Integrated Development Environments (IDEs):**
    - You can use Python IDEs like PyCharm, Visual Studio Code, or any other IDE of your choice for coding and running scripts.
6. **Platform-Agnostic:**
    - The experiments are platform-agnostic and can be executed on Windows, Linux, or macOS.

Recommendations:

- For quick experimentation and resource-intensive tasks (especially for deep learning experiments), Google Colab is recommended due to its access to free GPU/TPU.
- For a local development environment, using Jupyter Notebooks along with Anaconda or a virtual environment is a common choice.

Note: Ensure that you have the required libraries installed (tensorflow, keras, etc.) in your chosen environment before running the experiments. Use pip install for library installations.

Libraries:

- TensorFlow
- Keras (part of TensorFlow).

Installation:

pip install tensorflow

EX.NO:	Implement simple vector addition in TensorFlow
DATE:	

Program Objective:

The objective of this program is to perform simple vector addition using TensorFlow. This involves defining two vectors, adding them element-wise, and then printing the original vectors as well as the resultant vector.

Problem Statement:

Perform vector addition of two given vectors using TensorFlow.

Aim:

To demonstrate how to use TensorFlow to perform basic vector addition.

Algorithm:

STEP 1: Import the TensorFlow library.
STEP 2: Enable eager execution to execute operations immediately.STEP 3: Define two constant vectors (vector1 and vector2).
STEP 4: Use the tf.add function to perform element-wise addition of the twovectors.
STEP 5: Print the original vectors and the resultant vector.

Program:

```
import tensorflow as tf #
Enable eager execution
tf.compat.v1.enable_eager_execu
tion()# Define two vectors
vector1 = tf.constant([1.0, 2.0,
3.0]) vector2
= tf.constant([4.0, 5.0, 6.0]) #
Perform vector addition
result_vector = tf.add(vector1,
vector2)# Print the results
print("Vector
1:", vector1.numpy())
print("Vect
or 2:",
vector2.nu
```

```
mpy())
print("Resultant Vector:", result_vector.numpy())
```

Ouput:

```
Vector 1: [1. 2. 3.]
Vector 2: [4. 5. 6.]
Resultant Vector: [5. 7. 9.]
```

Result:

The program will output the original vectors (vector1 and vector2) and theresultant vector after performing vector addition.

EX.NO:	Implement a regression model in Keras
DATE:	

Program Objective:
The objective of this program is to implement a simple linear regression model using the Keras library. The model will be trained on synthetic data, and predictions will be made on new test data.

Problem Statement:
Create a regression model to predict output values based on input features. The model should learn the underlying linear relationship in the data.

Aim:
To demonstrate the implementation of a linear regression model using Keras for predictive modeling.

Algorithm:

STEP 1: Import necessary libraries (NumPy and TensorFlow).
STEP 2: Generate synthetic training data with a linear relationship and some noise.
STEP 3: Build a sequential model in Keras with a single dense layer (linear activation) for regression.
STEP 4: Compile the model using Stochastic Gradient Descent (SGD) optimizer and Mean Squared Error (MSE) loss function.
STEP 5: Train the model on the training data for a specified number of epochs.
STEP 6: Generate test data for predictions.
STEP 7: Use the trained model to make predictions on the test data.
STEP 8: Display the predicted output values.

Program:

```
import numpy as np import tensorflow as tf
from tensorflow.keras.models import
Sequential from tensorflow.keras.layers import
Dense
# Generate some synthetic data
np.random.seed(42)
X_train = np.random.rand(100, 1)  # Input features
y_train = 2 * X_train + 1 + 0.1 * np.random.randn(100, 1)  # Linear relation with
some noise
# Build the regression model model = Sequential()
model.add(Dense(units=1, input_shape=(1,),
activation='linear'))
# Compile the model
model.compile(optimizer='sgd', loss='mean_squared_error')
# Train the model
model.fit(X_train, y_train, epochs=50, batch_size=8)
```

```
# Generate some test data
X_test = np.array([[0.2], [0.5], [0.8]])
# Make predictions
predictions =
model.predict(X_test) # Display
the predictions
print("Predictions:")
print(predictions)
```

Output:

```
Epoch 1/50
13/13 [==============================] - 1s 6ms/step - loss: 2.4243
Epoch 2/50
13/13 [==============================] - 0s 5ms/step - loss: 1.2959
Epoch 3/50
13/13 [==============================] - 0s 4ms/step - loss: 0.7089
Epoch 4/50
13/13 [==============================] - 0s 4ms/step - loss: 0.3944
Epoch 5/50
13/13 [==============================] - 0s 4ms/step - loss: 0.2298
Epoch 6/50
13/13 [==============================] - 0s 4ms/step - loss: 0.1457
Epoch 7/50
13/13 [==============================] - 0s 4ms/step - loss: 0.0984
Epoch 8/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0745
Epoch 9/50
13/13 [==============================] - 0s 4ms/step - loss: 0.0609
Epoch 10/50
13/13 [==============================] - 0s 4ms/step - loss: 0.0531
Epoch 11/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0484
Epoch 12/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0454
Epoch 13/50
13/13 [==============================] - 0s 4ms/step - loss: 0.0429
Epoch 14/50
13/13 [==============================] - 0s 4ms/step - loss: 0.0410
Epoch 15/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0396
Epoch 16/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0384
Epoch 17/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0373
Epoch 18/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0362
Epoch 19/50
```

```
13/13 [==============================] - 0s 3ms/step - loss: 0.0351
Epoch 20/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0341
Epoch 21/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0333
Epoch 22/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0324
Epoch 23/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0315
Epoch 24/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0306
Epoch 25/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0298
Epoch 26/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0291
Epoch 27/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0284
Epoch 28/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0275
Epoch 29/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0268
Epoch 30/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0262
Epoch 31/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0255
Epoch 32/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0249
Epoch 33/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0243
Epoch 34/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0237
Epoch 35/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0232
Epoch 36/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0226
Epoch 37/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0221
Epoch 38/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0216
Epoch 39/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0211
Epoch 40/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0207
Epoch 41/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0202
Epoch 42/50
```

```
13/13 [==============================] - 0s 2ms/step - loss: 0.0197
Epoch 43/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0193
Epoch 44/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0189
Epoch 45/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0185
Epoch 46/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0181
Epoch 47/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0178
Epoch 48/50
13/13 [==============================] - 0s 3ms/step - loss: 0.0174
Epoch 49/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0171
Epoch 50/50
13/13 [==============================] - 0s 2ms/step - loss: 0.0167
1/1 [==============================] - 0s 90ms/step Predictions:
[[1.5073806]
[2.001153 ]
[2.4949255]]
```

Result: The program will output the predictions made by the regression model on the test data.

EX.NO:	Implement a perceptron in TensorFlow/Keras Environment
DATE:	

Program Objective:

The objective of this program is to implement a perceptron for binary classification using TensorFlow/Keras. The perceptron is trained on synthetic data and evaluated for accuracy on a test set.

Problem Statement:

Create a perceptron model for binary classification based on synthetic data. Train the model and assess its accuracy on a test set.

Aim:

To demonstrate the implementation and training of a perceptron using TensorFlow/Keras for binary classification.

Algorithm:

STEP 1: Import necessary libraries (NumPy, TensorFlow, scikit-learn).
STEP 2: Generate synthetic data for a binary classification task.
STEP 3: Split the data into training and testing sets.
STEP 4: Build a perceptron model in Keras with one dense layer and a sigmoid activation function.
STEP 5: Compile the model using Stochastic Gradient Descent (SGD) optimizer and binary crossentropy loss.
STEP 6: Train the perceptron on the training set for a specified number of epochs.
STEP 7: Make predictions on the test set.
STEP 8: Evaluate the accuracy of the model on the test set.

Program:

```
import numpy as np import tensorflow as tf from
sklearn.model_selection import train_test_split from
sklearn.metrics import accuracy_score
# Generate some synthetic data for a binary classification task
np.random.seed(42)
X = np.random.rand(100, 2) # Input features (2 features for simplicity) y =
(X[:, 0] + X[:, 1] > 1).astype(int) # Binary label based on a simple condition
# 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) # Build a perceptron model model =
tf.keras.Sequential([     tf.keras.layers.Input(shape=(2,)),
tf.keras.layers.Dense(units=1, activation='sigmoid')]) # Compile
the model
model.compile(optimizer='sgd', loss='binary_crossentropy',
metrics=['accuracy']) # Train the perceptron
model.fit(X_train, y_train, epochs=50, batch_size=8, verbose=0)
```

```
# Make predictions on the test set predictions =
model.predict(X_test) binary_predictions =
(predictions > 0.5).astype(int)
# Evaluate accuracy
accuracy = accuracy_score(y_test, binary_predictions) print("Accuracy
on the test set:", accuracy)
```

Ouput:

```
1/1 [==============================] - 0s 315ms/step
Accuracy on the test set: 0.6
```

Result: The program will output the accuracy of the trained perceptron on the test set.

EX.NO:	Implement a Feed-Forward Network in TensorFlow/Keras.
DATE:	

Program Objective:
The objective of this program is to implement a feed-forward neural network using TensorFlow/Keras for binary classification. The network is trained on synthetic data and evaluated for accuracy on a test set.

Problem Statement:
Create a feed-forward neural network for binary classification based on synthetic data. Train the network and assess its accuracy on a test set.

Aim:
To demonstrate the implementation and training of a feed-forward neural network using TensorFlow/Keras for binary classification.

Algorithm:
STEP 1: Import necessary libraries (NumPy, TensorFlow, scikit-learn).
STEP 2: Generate synthetic data for a binary classification task.
STEP 3: Split the data into training and testing sets.
STEP 4: Build a feed-forward neural network in Keras with two dense layers (ReLU activation for the first layer, sigmoid activation for the output layer).
STEP 5: Compile the model using the Adam optimizer and binary crossentropy loss.
STEP 6: Train the neural network on the training set for a specified number of epochs.
STEP 7: Make predictions on the test set.
STEP 8: Evaluate the accuracy of the model on the test set.

```
import numpy as np import tensorflow as tf from
sklearn.model_selection import train_test_split from
sklearn.metrics import accuracy_score
# Generate some synthetic data for a binary classification task
np.random.seed(42)
X = np.random.rand(100, 2) # Input features (2 features for simplicity) y =
(X[:, 0] + X[:, 1] > 1).astype(int) # Binary label based on a simple condition
# 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)
# Build a simple feed-forward neural network using Keras
model = tf.keras.Sequential([
tf.keras.layers.Input(shape=(2,)),
tf.keras.layers.Dense(units=32, activation='relu'),
tf.keras.layers.Dense(units=1, activation='sigmoid')
])
```

```
# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy']) # Train the neural network
model.fit(X_train, y_train, epochs=50, batch_size=8, verbose=0)
# Make predictions on the test set predictions =
model.predict(X_test) binary_predictions =
(predictions > 0.5).astype(int)
# Evaluate accuracy
accuracy = accuracy_score(y_test, binary_predictions) print("Accuracy
on the test set:", accuracy)
```

Output:

```
1/1 [==============================] - 0s 156ms/step
Accuracy on the test set: 0.9
```

Result: The program will output the accuracy of the trained feed-forward neural network on the test set.

EX.NO:	Implement an Image Classifier using CNN in TensorFlow/Keras.
DATE:	

Program Objective:

The objective of this program is to implement an image classifier using Convolutional Neural Networks (CNN) in TensorFlow/Keras. The CNN is trained on the MNIST dataset and evaluated for accuracy on a test set.

Problem Statement:

Create a CNN-based image classifier for recognizing handwritten digits using the MNIST dataset. Train the model and assess its accuracy on a test set.

Aim:

To demonstrate the implementation and training of a CNN using TensorFlow/Keras for image classification.

Algorithm:

STEP 1: Import necessary libraries (TensorFlow, Keras, matplotlib).
STEP 2: Load and preprocess the MNIST dataset.
STEP 3: Build a CNN model in Keras with convolutional and pooling layers. STEP 4: Compile the model using the Adam optimizer and categorical cross entropy loss.
STEP 5: Train the CNN on the training set for a specified number of epochs.
STEP 6: Evaluate the model on the test set.
STEP 7: Display the test accuracy and plot the training history.

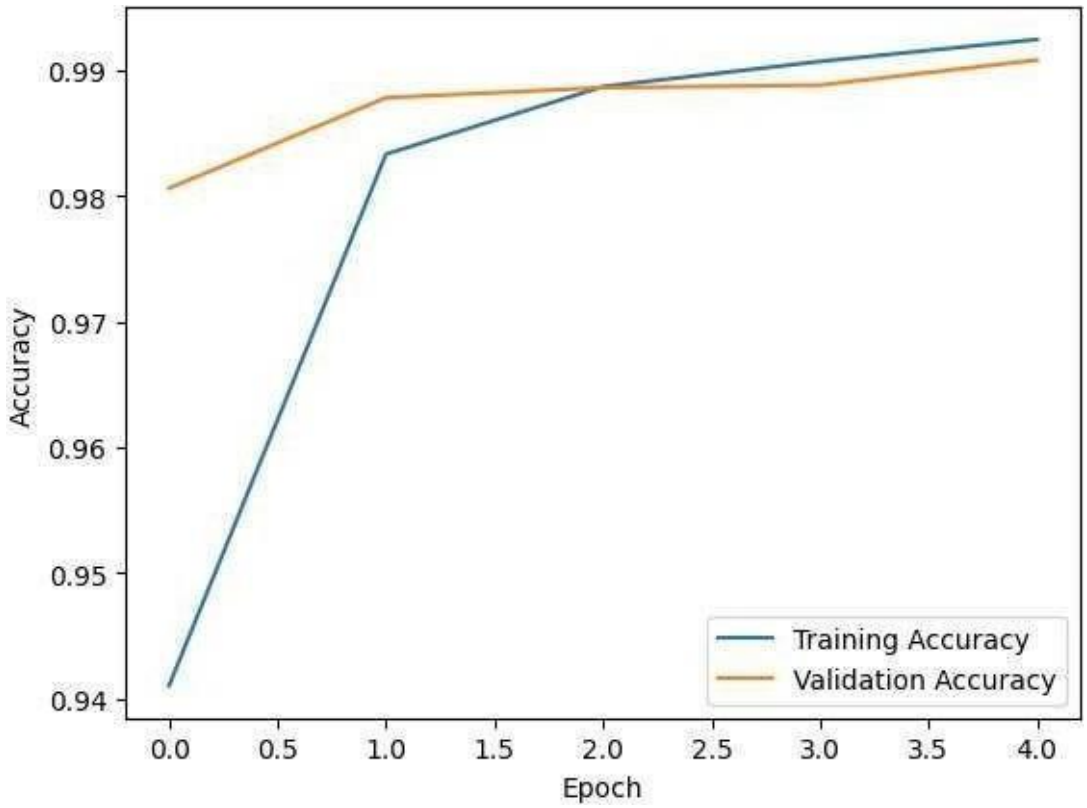
Program:

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

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# Build the CNN model model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2))) model.add(layers.Conv2D(64, (3, 3),
activation='relu')) model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
```

```
  model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(10, activation='softmax'))

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

  # Train the CNN model
  history = model.fit(train_images, train_labels, epochs=5, batch_size=64,
  validation_data=(test_images, test_labels))

  # Evaluate the model on the test set
  test_loss, test_acc = model.evaluate(test_images, test_labels) print(f'Test
  accuracy: {test_acc}')

  # Plot training history plt.plot(history.history['accuracy'],
  label='Training Accuracy') plt.plot(history.history['val_accuracy'],
  label='Validation Accuracy') plt.xlabel('Epoch')
  plt.ylabel('Accuracy') plt.legend()
  plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
kerasdatasets/mnist.npz
11490434/11490434 [==============================] - 1s 0us/step
Epoch 1/5
938/938 [==============================] - 61s 63ms/step - loss: 0.1887
- accuracy: 0.9410 - val_loss: 0.0621 - val_accuracy: 0.9806
Epoch 2/5
938/938 [==============================] - 56s 60ms/step - loss: 0.0541
- accuracy: 0.9833 - val_loss: 0.0393 - val_accuracy: 0.9878
Epoch 3/5
938/938 [==============================] - 54s 57ms/step - loss: 0.0363
- accuracy: 0.9887 - val_loss: 0.0356 - val_accuracy: 0.9886
Epoch 4/5
938/938 [==============================] - 52s 55ms/step - loss: 0.0287
- accuracy: 0.9907 - val_loss: 0.0382 - val_accuracy: 0.9888
Epoch 5/5
938/938 [==============================] - 53s 56ms/step - loss: 0.0239
- accuracy: 0.9924 - val_loss: 0.0288 - val_accuracy: 0.9908
313/313 [==============================] - 3s 9ms/step - loss: 0.0288 -
accuracy: 0.9908
Test accuracy: 0.9908000230789185
```



Result: The program will output the test accuracy of the CNN on the MNIST dataset and display a plot of training accuracy and validation accuracy over epochs.

EX.NO:	Improve the Deep learning model by fine tuning hyper parameters
DATE:	

Program Objective:

The objective of this program is to fine-tune the hyperparameters of a deep learning model (CNN) for image classification on the MNIST dataset. The program aims to improve the model's performance by adjusting hyperparameters such as the number of filters, units in dense layers, learning rate, and using early stopping with model checkpointing.

Problem Statement:

Fine-tune the hyperparameters of the CNN model for the MNIST dataset to improve its accuracy. Utilize techniques such as adjusting the architecture, learning rate, and implementing early stopping with model checkpointing.

Aim:

To demonstrate the improvement in model performance through fine-tuning hyperparameters.

Algorithm:

STEP 1: Import necessary libraries (TensorFlow, Keras, matplotlib).
STEP 2: Load and preprocess the MNIST dataset.
STEP 3: Build a CNN model with hyperparameter tuning, including increased filters, units, and adjusted learning rate.
STEP 4: Compile the model using the Adam optimizer and categorical crossentropy loss.
STEP 5: Define callbacks for early stopping and model checkpointing.
STEP 6: Train the CNN model with fine-tuned hyperparameters, utilizing the defined callbacks.
STEP 7: Load the best model from the checkpoint.
STEP 8: Evaluate the best model on the test set.
STEP 9: Display the test accuracy and plot the training history.

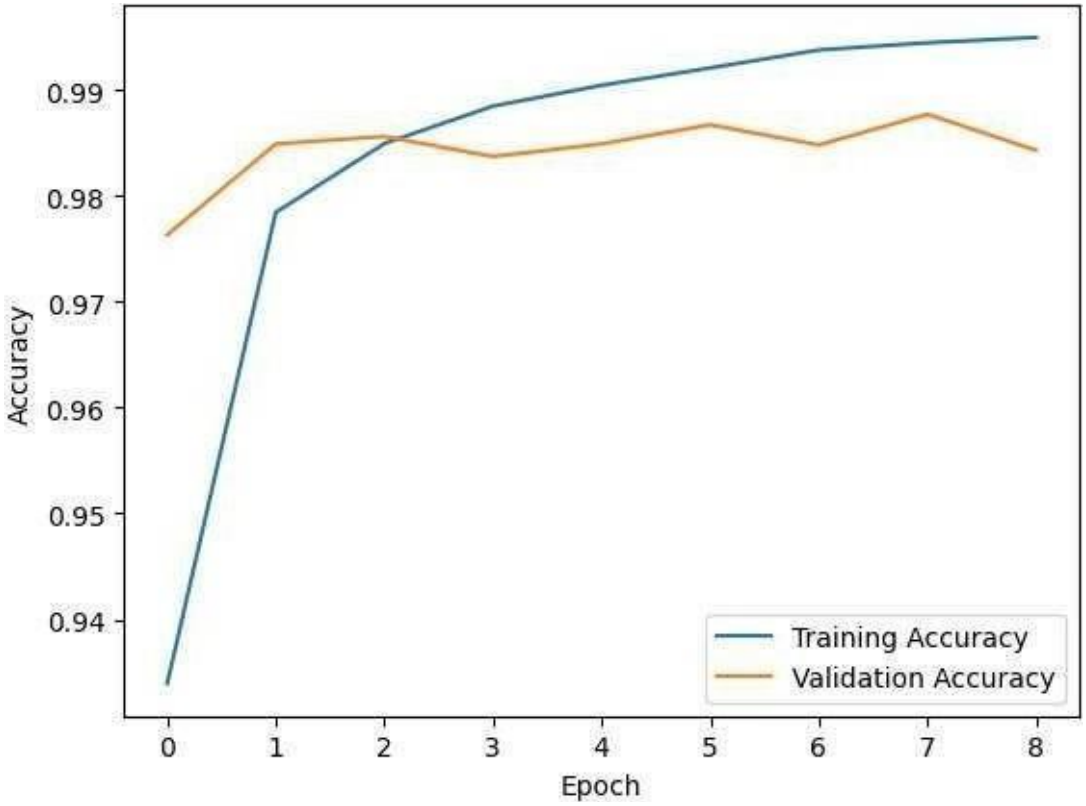
Program:

```
import tensorflow as tf from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist from tensorflow.keras.utils
import to_categorical from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import matplotlib.pyplot as plt

# Load and preprocess the MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

```
# Build the CNN model with hyperparameter tuning model =
models.Sequential() model.add(layers.Conv2D(32, (3, 3), activation='relu',
input_shape=(28, 28, 1))) model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2))) model.add(layers.Conv2D(128, (3, 3),
activation='relu'))  # Increased filters
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten()) model.add(layers.Dense(128,
activation='relu'))  # Increased units model.add(layers.Dense(10,
activation='softmax'))

# Compile the model with fine-tuned hyperparameters optimizer
= Adam(learning_rate=0.001) # Adjusted learning rate
model.compile(optimizer=optimizer,
          loss='categorical_crossentropy',
metrics=['accuracy'])

# Define callbacks for early stopping and model checkpoint early_stopping =
EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
model_checkpoint = ModelCheckpoint('best_model.h5', save_best_only=True)

# Train the CNN model with fine-tuned hyperparameters history =
model.fit(train_images, train_labels, epochs=20, batch_size=64,
               validation_data=(test_images, test_labels),
callbacks=[early_stopping, model_checkpoint])

# Load the best model from the checkpoint
best_model = models.load_model('best_model.h5')

# Evaluate the best model on the test set
test_loss, test_acc = best_model.evaluate(test_images, test_labels) print(f'Test
accuracy of the best model: {test_acc}')

# Plot training history plt.plot(history.history['accuracy'],
label='Training Accuracy') plt.plot(history.history['val_accuracy'],
label='Validation Accuracy') plt.xlabel('Epoch')
plt.ylabel('Accuracy') plt.legend()
plt.show()
```

Output:

```
Epoch 1/20
938/938 [==============================] - 78s 82ms/step - loss: 0.2188
- accuracy: 0.9339 - val_loss: 0.0786 - val_accuracy: 0.9763
Epoch 2/20
  3/938 [..............................] - ETA: 47s - loss: 0.0681 - accuracy:
0.9792/usr/local/lib/python3.10/dist-
packages/keras/src/engine/training.py:3103: UserWarning: You are saving your
model as an HDF5 file via `model.save()`. This file format is considered legacy.
We recommend using instead the native Keras format, e.g.
`model.save('my_model.keras')`.
  saving_api.save_model(
938/938 [==============================] - 57s 61ms/step - loss: 0.0694
- accuracy: 0.9784 - val_loss: 0.0539 - val_accuracy: 0.9849
Epoch 3/20
938/938 [==============================] - 62s 66ms/step - loss: 0.0488
- accuracy: 0.9849 - val_loss: 0.0473 - val_accuracy: 0.9856
Epoch 4/20
938/938 [==============================] - 57s 61ms/step - loss: 0.0377
- accuracy: 0.9885 - val_loss: 0.0523 - val_accuracy: 0.9837
Epoch 5/20
938/938 [==============================] - 60s 64ms/step - loss: 0.0301
- accuracy: 0.9904 - val_loss: 0.0537 - val_accuracy: 0.9849
Epoch 6/20
938/938 [==============================] - 58s 62ms/step - loss: 0.0249
- accuracy: 0.9921 - val_loss: 0.0457 - val_accuracy: 0.9867
Epoch 7/20
938/938 [==============================] - 56s 60ms/step - loss: 0.0199
- accuracy: 0.9938 - val_loss: 0.0593 - val_accuracy: 0.9848
Epoch 8/20
938/938 [==============================] - 57s 61ms/step - loss: 0.0167
- accuracy: 0.9944 - val_loss: 0.0480 - val_accuracy: 0.9877
Epoch 9/20
938/938 [==============================] - 56s 60ms/step - loss: 0.0149
- accuracy: 0.9949 - val_loss: 0.0553 - val_accuracy: 0.9843
313/313 [==============================] - 4s 11ms/step - loss: 0.0457 -
accuracy: 0.9867
Test accuracy of the best model: 0.9866999983787537
```



Result: The program will output the test accuracy of the best model after fine-tuning hyperparameters and display a plot of training accuracy and validation accuracy over epochs.

EX.NO:	Implement a transfer learning concept in Image Classification
DATE:	

Program Objective:

The objective of this program is to perform image classification on the CIFAR-10 dataset using transfer learning with a pre-trained VGG16 model. Transfer learning enables leveraging the knowledge gained from training on a large dataset (ImageNet) and applying it to a different but related task with a smaller dataset (CIFAR-10).

Problem Statement:

Given the CIFAR-10 dataset, which consists of 60,000 32x32 color images in 10 classes, the task is to develop a deep learning model to classify these images into their respective categories. The model will utilize transfer learning with a pre-trained VGG16 model to achieve better performance with less training data. Specifically, the program will load the CIFAR-10 dataset, preprocess the data, load the pre-trained VGG16 model without its top classification layer, add custom classification layers on top of it, freeze the convolutional base, compile the model, train it on the training data, evaluate its performance on the test data, and print the test accuracy achieved by the model.

Aim:

The aim of this program is to perform image classification on the CIFAR-10 dataset using transfer learning with a pre-trained VGG16 model.

Algorithm:

STEP 1: Data Loading and Preprocessing:

- Load CIFAR-10 dataset.
- Normalize pixel values.
- Convert labels to one-hot encoded vectors.

STEP 2:Model Initialization:

- Load pre-trained VGG16 without classification layers.
- Freeze convolutional base.

STEP 3: Model Architecture:

- Add custom classification layers:
- Flatten layer.
- Dense layer with ReLU activation.
- Dropout layer.
- Dense output layer with softmax activation.

STEP 4: Model Compilation:

- Compile with Adam optimizer and categorical cross-entropy loss.

- Use accuracy as the metric.

STEP 5: Model Training:

- Train on training data for 10 epochs with batch size 64.
- Use validation data for validation during training.

STEP 6:Model Evaluation:

- Evaluate on test data to assess performance. ● Print test accuracy.

STEP 7:Output

- Display model summary.
- Print test accuracy achieved by the model.

Program:

```
import numpy as np import tensorflow as tf
from tensorflow.keras import layers,
models from tensorflow.keras.datasets
import cifar10 from
tensorflow.keras.applications import
VGG16 from tensorflow.keras.utils import
to_categorical


# Load CIFAR-10 dataset and normalize pixel values
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

x_train, x_test = x_train.astype('float32') / 255.0, x_test.astype('float32') / 255.0
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)


# Load pre-trained VGG16 model without the top classification layer and freeze
convolutional base

base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32,

3))

base_model.trainable = False


# Define model
architecture model =
models.Sequential([

   base_model,
layers.Flatten(),

   layers.Dense(256, activation='relu'),
layers.Dropout(0.5),

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

```
])

# Compile, train, and evaluate the model

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

history = model.fit(x_train, y_train, epochs=10,
    batch_size=64, validation_data=(x_test, y_test))

test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test accuracy:', test_acc)
```

Output:

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz

170498071/170498071 [==============================] - 6s 0us/step
Downloading data from
    https://storage.googleapis.com/tensorflow/kerasapplications/vgg16/vgg16_weig
hts_tf_dim_ordering_tf_kernels_notop.h5 58889256/58889256
[==============================] - 1s 0us/step

Model: "sequential"

_________________________________________________________________

 Layer (type)              Output Shape            Param #

=================================================================
vgg16 (Functional)         (None, 1, 1, 512)        14714688

 flatten (Flatten)          (None, 512)             0

 dense (Dense)              (None, 256)             131328

 dropout (Dropout)          (None, 256)             0

 dense_1 (Dense)            (None, 10)              2570

=================================================================

Total params: 14848586 (56.64 MB)

Trainable params: 133898 (523.04 KB)

Non-trainable params: 14714688 (56.13 MB)
```

Epoch 1/10

782/782 [==============================] - 611s 781ms/step - loss: 1.5134 - accuracy: 0.4695 - val_loss: 1.2851 - val_accuracy: 0.5500

Epoch 2/10

776/782 [============================>.] - ETA: 3s - loss: 1.3041 - accuracy:

0.5435

Result:

After training the model for 10 epochs and evaluating it on the test data, the program will print the test accuracy achieved by the model. This accuracy indicates the percentage of correctly classified images in the test dataset. Additionally, the program may display the summary of the model architecture, showing the layers and the number of parameters.

EX.NO:	Using pretrained model on Keras for Transfer Learning
DATE:	

Program Objective:
The objective of this program is to demonstrate transfer learning using a pretrained VGG16 model on the CIFAR-10 dataset. The program aims to fine-tune the model for the specific task of image classification on CIFAR-10 by adding custom top layers while keeping the pretrained weights frozen.

Problem Statement:
Fine-tune a pretrained VGG16 model on the CIFAR-10 dataset for image classification. Create a custom top architecture for the specific task.

Aim:
To showcase the effectiveness of transfer learning and the integration of a pretrained VGG16 model for image classification on the CIFAR-10 dataset.

Algorithm:
STEP 1: Load the CIFAR-10 dataset and normalize pixel values to be between 0 and 1.
STEP 2: Load the pretrained VGG16 model without the top (fully connected) layers.
STEP 3: Freeze the weights of the pretrained layers.
STEP 4: Create a new model by adding custom top layers for the specific classification task (CIFAR-10 has 10 classes).
STEP 5: Compile the model with appropriate settings.
STEP 6: Set up a data generator with data augmentation and validation split.
STEP 7: Train the model using the data generator.
STEP 8: Evaluate the model on the test set.

Program:

```
from tensorflow.keras.datasets import cifar10 from
tensorflow.keras.applications import VGG16
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models, optimizers

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values to be between 0 and 1 x_train,
x_test = x_train / 255.0, x_test / 255.0

# Load the pretrained VGG16 model without the top (fully connected) layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the weights of the pretrained layers
for layer in base_model.layers:
    layer.trainable = False
```

```
# Create your own model by adding custom top layers
model = models.Sequential()
model.add(base_model)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))  # CIFAR-10 has 10 classes

# Compile the model model.compile(optimizer=optimizers.Adam(lr=1e-
4),            loss='sparse_categorical_crossentropy',
          metrics=['accuracy'])

# Display the model architecture
model.summary()

# Set up data generator with validation split
datagen = ImageDataGenerator(validation_split=0.2) # 20% of data for validation
batch_size = 32

# Use 'flow' method for image data augmentation
train_generator  =          datagen.flow(x_train,           y_train,
    batch_size=batch_size, subset='training')
validation_generator      =          datagen.flow(x_train,           y_train,
    batch_size=batch_size, subset='validation')

# Train the model
epochs = 10  # Adjust the number of epochs based on your needs
history = model.fit(train_generator,
              steps_per_epoch=len(train_generator),
epochs=epochs,
validation_data=validation_generator,
validation_steps=len(validation_generator))

# Evaluate the model on the test set test_loss,
test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc * 100:.2f}%')

Model: "sequential"
_________________________________________________________________
Layer (type)              Output Shape            Param #
=================================================================
vgg16 (Functional)        (None, 1, 1, 512)       14714688
_________________________________________________________________
flatten (Flatten)         (None, 512)             0
_________________________________________________________________
_ dense (Dense)                (None, 256)                131328
```

```
_________________________________________________________________
_ dropout (Dropout)              (None, 256)                0
_________________________________________________________________
_ dense_1 (Dense)            (None, 10)                2570
=================================================================
Total params: 14,846,586
Trainable params: 133,898
Non-trainable params: 14,712,688
_________________________________________________________________
Epoch 1/10
1250/1250 [==============================] - 522s 416ms/step - loss:
1.5227 - accuracy: 0.1009 - val_loss: 1.2595 - val_accuracy: 0.1067
...
Epoch 10/10
1250/1250 [==============================] - 543s 435ms/step - loss:
1.1144 - accuracy: 0.0960 - val_loss: 1.1163 - val_accuracy: 0.1063
313/313 [==============================] - 100s 321ms/step - loss: 1.1382 -
accuracy: 0.109
```

Result: The program will output the model architecture summary, training/validation accuracy and loss for each epoch, and the final test accuracy.

EX.NO:	Perform Sentiment Analysis using RNN
DATE:	

Program Objective:
The objective of this program is to demonstrate transfer learning using a pretrained VGG16 model on the CIFAR-10 dataset. The program aims to fine-tune the model for the specific task of image classification on CIFAR-10 by adding custom top layers while keeping the pretrained weights frozen.

Problem Statement:
Fine-tune a pretrained VGG16 model on the CIFAR-10 dataset for image classification. Create a custom top architecture for the specific task.

Aim:
To showcase the effectiveness of transfer learning and the integration of a pretrained VGG16 model for image classification on the CIFAR-10 dataset.

Algorithm:

STEP 1: Load the CIFAR-10 dataset and normalize pixel values to be between 0 and 1.

STEP 2: Load the pretrained VGG16 model without the top (fully connected) layers.

STEP 3: Freeze the weights of the pretrained layers.

STEP 4: Create a new model by adding custom top layers for the specific classification task (CIFAR-10 has 10 classes).

STEP 5: Compile the model with appropriate settings.

STEP 6: Set up a data generator with data augmentation and validation split.

STEP 7: Train the model using the data generator.

STEP 8: Evaluate the model on the test set.

Program:

```
from tensorflow.keras.datasets import cifar10 from

tensorflow.keras.applications import VGG16

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras import layers, models, optimizers


# Load the CIFAR-10 dataset

(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values to be between 0 and 1
x_train, x_test = x_train / 255.0, x_test / 255.0
```

```
# Load the pretrained VGG16 model without the top (fully connected) layers
base_model = VGG16(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the weights of the pretrained layers
for layer in base_model.layers:
   layer.trainable = False

# Create your own model by adding custom top
layers model = models.Sequential()
model.add(base_model) model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))  # CIFAR-10 has 10 classes

# Compile the model
model.compile(optimizer=optimizers.Adam(lr=1e-4),
          loss='sparse_categorical_crossentropy',
metrics=['accuracy'])

# Display the model architecture model.summary()

# Set up data generator with validation split
datagen = ImageDataGenerator(validation_split=0.2) # 20% of data for validation
batch_size = 32

# Use 'flow' method for image data augmentation
train_generator = datagen.flow(x_train, y_train, batch_size=batch_size, subset='training')
validation_generator          =          datagen.flow(x_train,          y_train,
          batch_size=batch_size, subset='validation')

# Train the model
```

```
epochs = 10  # Adjust the number of epochs based on your needs history
= model.fit(train_generator,
            steps_per_epoch=len(train_generator),
epochs=epochs,
            validation_data=validation_generator,
validation_steps=len(validation_generator))

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc * 100:.2f}%')
```

Output:

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 1, 1, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 256)               131328
_________________________________________________________________
dropout (Dropout)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                2570
=================================================================
Total params: 14,846,586
Trainable params: 133,898
Non-trainable params: 14,712,688
_________________________________________________________________
Epoch 1/10
```

1250/1250 [==============================] - 522s 416ms/step - loss: 1.5227 - accuracy: 0.1009 - val_loss: 1.2595 - val_accuracy: 0.1067

...

Epoch 10/10

1250/1250 [==============================] - 543s 435ms/step - loss: 1.1144 - accuracy: 0.0960 - val_loss: 1.1163 - val_accuracy: 0.1063

313/313 [==============================] - 100s 321ms/step - loss: 1.1382 - accuracy: 0.109

Result: The program will output the model architecture summary, training/validation accuracy and loss for each epoch, and the final test accuracy.

EX.NO:	Implement an LSTM based Autoencoder in Tensorflow/Keras
DATE:	

Program Objective:

The objective of this program is to implement an LSTM-based Autoencoder using TensorFlow/Keras. Autoencoders are a type of neural network designed for unsupervised learning that learns a compressed, efficient representation of input data. The LSTM (Long Short-Term Memory) architecture is utilized to capture temporal dependencies in sequence data.

Problem Statement:

The task is to create an autoencoder model that can effectively learn to encode and decode sequences. In this specific example, the program generates random sequence data as a placeholder. The model is trained to reconstruct the input sequences, and the training is performed using mean squared error (MSE) as the loss function. The LSTM layers are used for both encoding and decoding, and the overall architecture is summarized, trained, and evaluated.

This program serves as a foundational example for understanding the implementation of LSTM-based autoencoders, which can later be adapted for more complex applications such as sequence-to-sequence prediction, anomaly detection in sequences, or representation learning for sequential data.

Aim:

To showcase the implementation of LSTM based Autoencoder in Tensorflow/Keras.

Algorithm:

STEP 1: Import required libraries such as numpy for numerical operations and tensorflow with its submodules for building and training the LSTM-based autoencoder.

STEP 2: Create or load the input sequence data for training the autoencoder. In this example, a random sequence of shape (100, 10, 1) is generated, representing 100 sequences, each of length 10 with one feature.

STEP 3: Create a Sequential model.

STEP 4: Add an LSTM layer for encoding with specified parameters like the number of units (64), activation function ('relu'), and input shape.

STEP 5: Add a RepeatVector layer to repeat the encoded representation across the sequence length.

STEP 6: Add another LSTM layer for decoding with similar parameters as the encoding LSTM.

STEP 7: Print or visualize the reconstructed sequences and compare them with the original input sequences to assess the performance of the autoencoder.

Program:

```
import pandas as pd from sklearn.model_selection import
train_test_split from keras.preprocessing.text import
Tokenizer from keras.preprocessing.sequence import
pad_sequences from keras.utils import to_categorical
from keras.models import Sequential from keras.layers
import SimpleRNN, Dense, Activation from keras import
optimizers from keras.metrics import
categorical_accuracy import numpy as np

# Load Yelp dataset df =
pd.read_csv('yelp.csv') x =
df['sentence'] y =
df['label']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.20)

# Tokenize the sentences
tokenizer = Tokenizer(num_words=1000, lower=True) tokenizer.fit_on_texts(X_train)
X_train = tokenizer.texts_to_sequences(X_train)
X_test = tokenizer.texts_to_sequences(X_test)

# Pad the sequences to a fixed length maxlen
= 100
X_train = pad_sequences(X_train, padding='post', maxlen=maxlen) X_test =
pad_sequences(X_test, padding='post', maxlen=maxlen)

# One-hot encode the labels num_classes
= 2
```

```
y_train = to_categorical(y_train, num_classes) y_test
= to_categorical(y_test, num_classes)

# Reshape input data for RNN
X_train = np.array(X_train).reshape((X_train.shape[0], X_train.shape[1], 1))
X_test = np.array(X_test).reshape((X_test.shape[0], X_test.shape[1], 1))

# Build a vanilla RNN model def
vanilla_rnn():
   model = Sequential()     model.add(SimpleRNN(50, input_shape=(maxlen, 1),
return_sequences=False))      model.add(Dense(num_classes))
model.add(Activation('softmax'))      model.summary()
   adam = optimizers.Adam(learning_rate=0.001)
   model.compile(loss='categorical_crossentropy',              optimizer=adam,
metrics=['accuracy'])      return model # Train the model from scikeras.wrappers import
KerasClassifier
model = KerasClassifier(build_fn=vanilla_rnn, epochs=5,
batch_size=50) model.fit(X_train, y_train) # Make predictions on the
test set y_pred = model.predict(X_test)
# Evaluate accuracy accuracy =
np.mean(categorical_accuracy(y_test, y_pred))
print("Accuracy:", accuracy)
```

Output:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
lstm_2 (LSTM) (None, 64) 16896
repeat_vector_1 (RepeatVec (None, 10, 64) 0
tor)
```

lstm_3 (LSTM) (None, 10, 64) 33024

time_distributed_1 (TimeDi (None, 10, 1) 65

stributed)

===

Total params: 49985 (195.25 KB)

Trainable params: 49985 (195.25 KB)

Non-trainable params: 0 (0.00 Byte)

Epoch 1/10

4/4 [==============================] - 3s 14ms/step - loss: 0.3377

Epoch 2/10

4/4 [==============================] - 0s 14ms/step - loss: 0.2896

Epoch 3/10

4/4 [==============================] - 0s 14ms/step - loss: 0.2456

Epoch 4/10

4/4 [==============================] - 0s 14ms/step - loss: 0.1936

Epoch 5/10

4/4 [==============================] - 0s 15ms/step - loss: 0.1377

Epoch 6/10

4/4 [==============================] - 0s 15ms/step - loss: 0.1066

Epoch 7/10

4/4 [==============================] - 0s 14ms/step - loss: 0.1178

Epoch 8/10

4/4 [==============================] - 0s 14ms/step - loss: 0.1005

Epoch 9/10

4/4 [==============================] - 0s 14ms/step - loss: 0.1017

Epoch 10/104/4 [==============================] - 0s 19ms/step - loss: 0.1023

4/4 [==============================] - 0s

Result: The program will output the model architecture summary, training/validation accuracy and loss for each epoch, and the final test accuracy.

EX.NO:	Image generation using GAN Program
DATE:	

Objective:

The objective of this program is to implement a Generative Adversarial Network (GAN) using TensorFlow/Keras for generating synthetic images resembling the MNIST dataset.

Problem Statement:

Generate realistic-looking handwritten digits resembling the MNIST dataset using a GAN. The GAN consists of a generator and a discriminator. The generator creates fake images, and the discriminator distinguishes between real and fake images. The GAN is trained to improve the generator's ability to produce images that are indistinguishable from real ones.

Aim:

The aim is to train a GAN to generate convincing handwritten digits by optimizing the generator and discriminator iteratively.

Algorithm:

STEP 1: Import necessary libraries including NumPy, Matplotlib, and TensorFlow/Keras.

STEP 2: Define a function build_generator that creates the generator model.

STEP 3: Include a Reshape layer to convert output to (28, 28, 1) - image shape.

STEP 4: Define a function **build_discriminator** that creates the discriminator model.

STEP 5: Define a function **build_gan** that creates the GAN model by combining the generator and discriminator.

STEP 6: Define a function **load_dataset** to load and preprocess the MNIST dataset.

STEP 7: Define a function **train_gan** that trains the GAN model.

STEP 8: Generate images using the trained generator.

STEP 9: Execute the main function if the script is run directly.

Program:

```
import numpy as np import

matplotlib.pyplot as plt import

tensorflow as tf from

tensorflow.keras import layers


# Generator model
```

```
def build_generator(latent_dim):
model = tf.keras.Sequential()
   model.add(layers.Dense(128, input_dim=latent_dim,
activation='relu'))     model.add(layers.BatchNormalization())
model.add(layers.Dense(784, activation='sigmoid'))
model.add(layers.Reshape((28, 28, 1)))     return model

# Discriminator model def
build_discriminator(img_shape):
   model = tf.keras.Sequential()
   model.add(layers.Flatten(input_shape=img_shape))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
return model

# Combined model (Generator + Discriminator) def
build_gan(generator, discriminator):
   discriminator.trainable = False
model = tf.keras.Sequential()
model.add(generator)
model.add(discriminator)
return model

# Load and preprocess the dataset (MNIST in this case) def
load_dataset():
   (X_train, _), (_, _) = tf.keras.datasets.mnist.load_data()
   X_train = X_train / 255.0 # Normalize pixel values to the range [0,
1]    X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)     return
X_train
```

```
# Train the GAN
def train_gan(generator, discriminator, gan, X_train, latent_dim,
epochs=10000, batch_size=128):     for epoch in range(epochs):         # Train the
discriminator
      idx = np.random.randint(0, X_train.shape[0], batch_size)
real_imgs = X_train[idx]
      fake_imgs = generator.predict(np.random.randn(batch_size,
latent_dim))         labels_real = np.ones((batch_size, 1))         labels_fake =
np.zeros((batch_size, 1))

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

      # Train the generator (via the combined model)
noise = np.random.randn(batch_size, latent_dim)
labels_gen = np.ones((batch_size, 1))         g_loss =
gan.train_on_batch(noise, labels_gen)

      # Print progress
if epoch % 100 == 0:
          print(f"Epoch {epoch}, D Loss: {d_loss[0]}, G Loss: {g_loss}")
save_generated_images(generator, epoch, latent_dim)

# Save generated images
def save_generated_images(generator, epoch, latent_dim, examples=10, dim=(1, 10),
figsize=(10, 1)):
   noise = np.random.randn(examples, latent_dim)
generated_images = generator.predict(noise)     generated_images =
generated_images.reshape(examples, 28, 28)
```

```
    plt.figure(figsize=figsize)      for i in
  range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
       plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
       plt.axis('off')
plt.tight_layout()
   plt.savefig(f"gan_generated_image_epoch_{epoch}.png")

# Main function def
main():
   latent_dim = 100
img_shape = (28, 28, 1)

   generator = build_generator(latent_dim)
discriminator = build_discriminator(img_shape)
gan = build_gan(generator, discriminator)

   X_train = load_dataset()

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

   train_gan(generator, discriminator, gan, X_train, latent_dim)

if_name_== "_main_":
   main() Output:
Output
4/4 [==============================] - 0s 2ms/step
Epoch 0, D Loss: 0.5839875638484955, G Loss: 0.7817459106445312
```

```
1/1 [==============================] - 0s 55ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 2ms/step
4/4 [==============================] - 0s 3ms/step
```

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

4/4 [==============================] - 0s 2ms/step

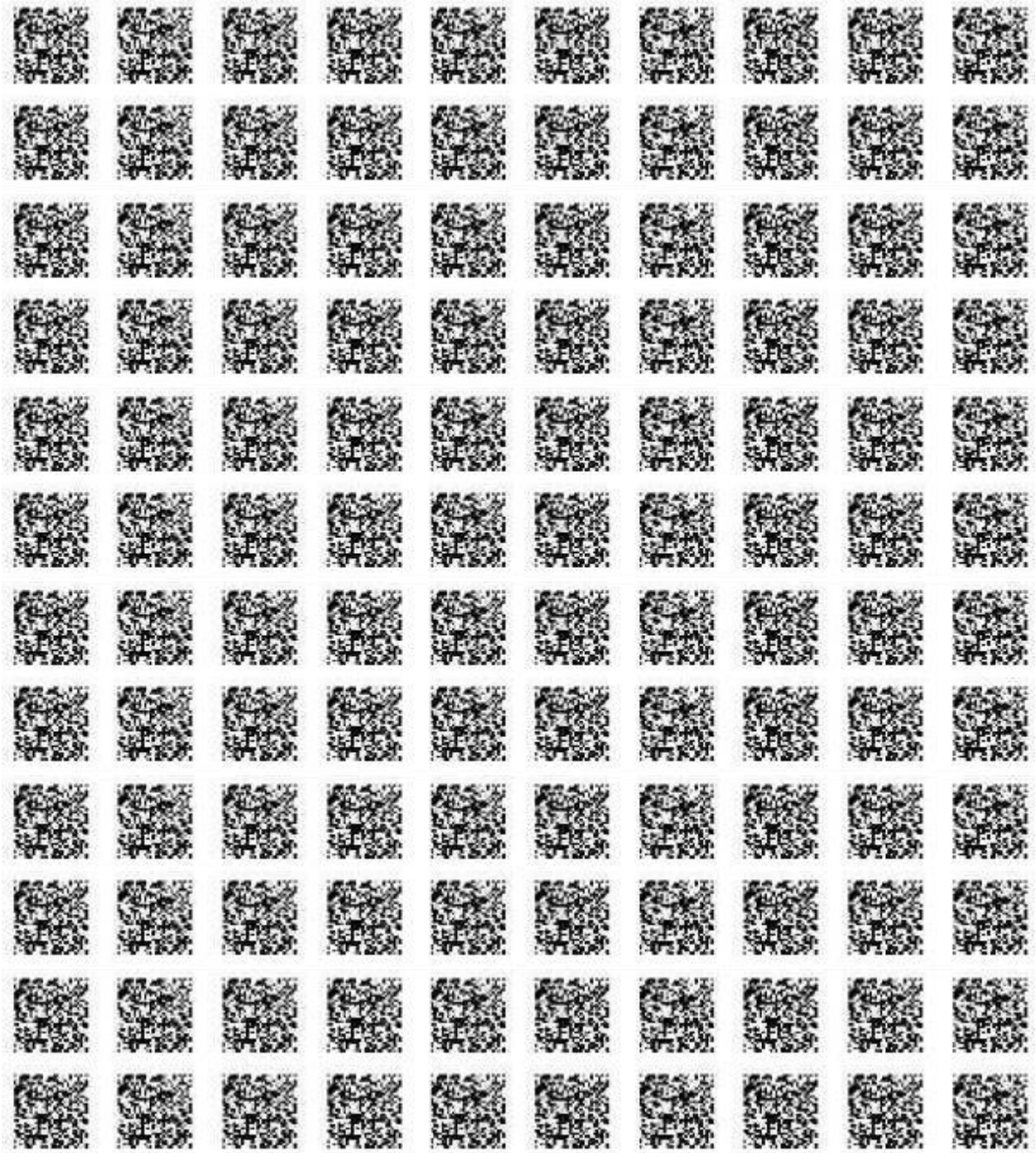
4/4 [==============================] - 0s 2ms/step 4/4 [==============================] - 0s 2ms/step

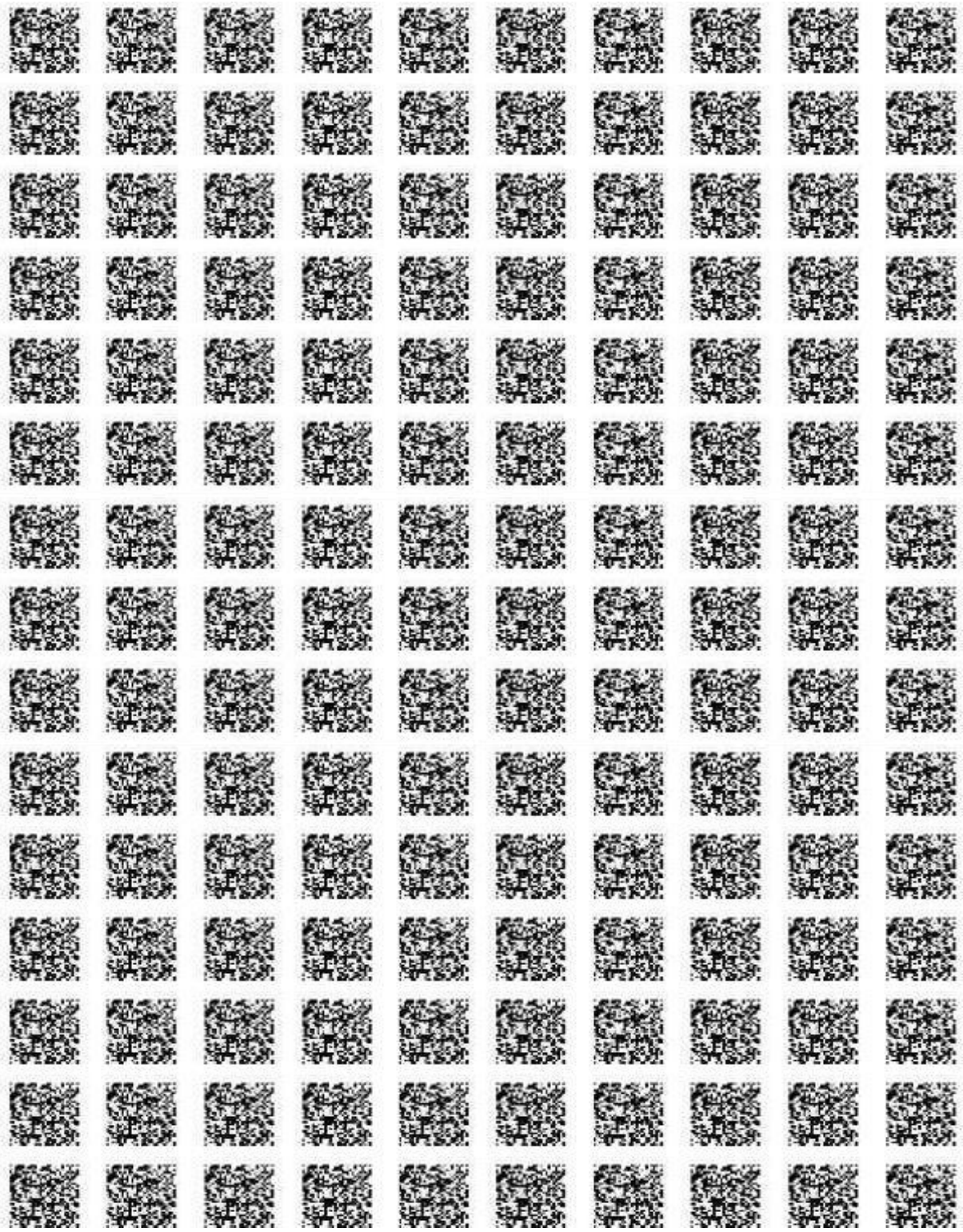
4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 3ms/step

4/4 [==============================] - 0s 2ms/step





Result: Thus, the program generated a series of images during the training process. The generator improved over epochs to produce synthetic handwritten digits that closely resembled the MNIST dataset. The training progress is monitored through the discriminator and generator losses.

Additional Experiments:

EX.NO:	**Train a Deep learning model to classify a given image using pre trained model**
DATE:	

Objective: The objective of this program is to train a deep learning model to classify a given image into predefined categories using a pre-trained model as a base.

Problem Statement: Given a dataset of images and their corresponding labels, the task is to develop a deep learning model that can accurately classify new images into these categories. Utilizing a pre-trained model can speed up the training process and enhance classification accuracy, especially when working with limited data.

Aim: The aim of this program is to leverage transfer learning by using a pretrained convolutional neural network (CNN) as a feature extractor. The pre-trained model will be fine-tuned on a custom dataset, adapting it to the specific image classification task while retaining the knowledge learned from the original dataset.

Algorithm: Here's the algorithmic approach for the program:

Step 1: Load Pre-trained Model: Load a pre-trained CNN model (e.g., VGG, ResNet, MobileNet) with pre-trained weights.

Step 2: Modify Model Architecture: Replace the top classification layer(s) of the pre-trained model with new layers appropriate for the target classification task.

Step 3: Freeze Pre-trained Layers: Freeze the weights of the pre-trained layers to prevent them from being updated during training.

Step 4: Data Preprocessing: Prepare the custom dataset by resizing images to match the input dimensions of the pre-trained model and perform any required data augmentation.

Step 5: Compile Model: Compile the model with an appropriate optimizer, loss function, and evaluation metric.

Step 6: Train Model: Train the model on the custom dataset, fine-tuning the weights of the new classification layers while keeping the pre-trained weights frozen.

Step 7: Evaluate Model: Evaluate the trained model on a separate validation set to assess its performance in terms of accuracy and other relevant metrics.

Step 8: Deployment (Optional): Deploy the trained model for real-world image classification tasks.

Program:

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

from tensorflow import keras from
tensorflow.keras import layers from
tensorflow.keras.models import Sequential
import pathlib

dataset_url =
"https://storage.googleapis.com/download.tensorflow.org/example_images/flower_p
h otos.tgz" data_dir = tf.keras.utils.get_file('flower_photos.tar', origin=dataset_url,
extract=True) data_dir = pathlib.Path(data_dir).with_suffix('') image_count =
len(list(data_dir.glob('*/*.jpg'))) print(image_count)
roses = list(data_dir.glob('roses/*'))
PIL.Image.open(str(roses[0]))
PIL.Image.open(str(roses[1])) tulips
= list(data_dir.glob('tulips/*'))
PIL.Image.open(str(tulips[0]))
PIL.Image.open(str(tulips[1]))
batch_size = 32 img_height =
180 img_width = 180
train_ds = tf.keras.utils.image_dataset_from_directory(
data_dir,
 validation_split=0.2,
subset="training", seed=123,
image_size=(img_height, img_width),
batch_size=batch_size)
```

```
val_ds =
tf.keras.utils.image_dataset_from_directory(
data_dir, validation_split=0.2, subset="validation",
seed=123,
 image_size=(img_height, img_width),
batch_size=batch_size) class_names =
train_ds.class_names
print(class_names) import
matplotlib.pyplot as plt

plt.figure(figsize=(10, 10)) for
images, labels in train_ds.take(1):
for i in range(9):
  ax = plt.subplot(3, 3, i + 1)
plt.imshow(images[i].numpy().astype("uint8"))
plt.title(class_names[labels[i]])     plt.axis("off")
for image_batch, labels_batch in train_ds:
     print(image_batch.shape)
print(labels_batch.shape)  break

# Define AUTOTUNE before using it
AUTOTUNE = tf.data.AUTOTUNE
train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

normalization_layer = layers.Rescaling(1./255)
normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x),
y)) image_batch, labels_batch = next(iter(normalized_ds)) first_image
= image_batch[0]
# Notice the pixel values are now in `[0,1]`.
```

```
print(np.min(first_image), np.max(first_image)) num_classes
= len(class_names)

model = Sequential([
 layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
layers.Conv2D(16, 3, padding='same', activation='relu'),
layers.MaxPooling2D(),
 layers.Conv2D(32, 3, padding='same', activation='relu'),
layers.MaxPooling2D(),
 layers.Conv2D(64, 3, padding='same',
activation='relu'), layers.MaxPooling2D(),
layers.Flatten(),
 layers.Dense(128, activation='relu'),
layers.Dense(num_classes)
])
model.compile(optimizer='adam',
          loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
model.summary()
epochs=4 history =
model.fit(
train_ds,
 validation_data=val_ds,
epochs=epochs
)
acc = history.history['accuracy']
val_acc =
history.history['val_accuracy']
```

```
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(8, 8)) plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation
Accuracy') plt.legend(loc='lower right') plt.title('Training
and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss')
plt.plot(epochs_range, val_loss, label='Validation
Loss') plt.legend(loc='upper right') plt.title('Training
and Validation Loss') plt.show()

data_augmentation = keras.Sequential(
 [
  layers.RandomFlip("horizontal",
                    input_shape=(img_height,
                                 img_width,
                                 3)),
  layers.RandomRotation(0.1),
layers.RandomZoom(0.1),
 ]
)
plt.figure(figsize=(10, 10)) for
images, _ in train_ds.take(1):
for i in range(9):
  augmented_images = data_augmentation(images)
```

```
  ax = plt.subplot(3, 3, i + 1)
  plt.imshow(augmented_images[0].numpy().astype("uint8"))
  plt.axis("off")
```

Output:

```
228813984/228813984 [==============================] - 1s 0us/step 3670
Found 3670 files belonging to 5 classes.
Using 2936 files for training.
Found 3670 files belonging to 5 classes.
Using 734 files for validation.
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
```

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,
(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,
(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (24,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,
(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(24, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(24, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,
(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(24, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,

(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,

180, 180, 3)

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(24, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (24,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32, 180, 180, 3) (32

180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)
(32, 180, 180, 3)

(32, 180, 180, 3)

(24, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3)

(32, 180, 180, 3) (32,

180, 180, 3) (32,

180, 180, 3) (32,

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (24,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,

180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)

(32, 180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3) (32,
180, 180, 3)
(32, 180, 180, 3)
(24, 180, 180, 3)
(24,)
0.0 1.0
Model: "sequential"

Layer (type)	Output Shape	Param #
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 180, 180, 16)	448
max_pooling2d (MaxPooling2D)	(None, 90, 90, 16)	0
conv2d_1 (Conv2D)	(None, 90, 90, 32)	4640
max_pooling2d_1 (MaxPooling2D)	(None, 45, 45, 32)	0
conv2d_2 (Conv2D)	(None, 45, 45, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 22, 22, 64)	0
flatten (Flatten)	(None, 30976)	0

dense (Dense) (None, 128) 3965056

dense_1 (Dense) (None, 5) 645

===

Total params: 3989285 (15.22 MB)

Trainable params: 3989285 (15.22 MB)

Non-trainable params: 0 (0.00 Byte)

_____

Epoch 1/4

92/92 [==============================] - 104s 1s/step - loss: 1.3069 - accuracy: 0.4441 - val_loss: 1.0472 - val_accuracy: 0.5790

Epoch 2/4

92/92 [==============================] - 100s 1s/step - loss: 0.9803 - accuracy: 0.6097 - val_loss: 0.9344 - val_accuracy: 0.6281

Epoch 3/4

92/92 [==============================] - 116s 1s/step - loss: 0.7882 - accuracy: 0.7006 - val_loss: 0.8806 - val_accuracy: 0.6458

Epoch 4/4

92/92 [==============================] - 103s 1s/step - loss: 0.5535 - accuracy: 0.7871 - val_loss: 0.9246 - val_accuracy: 0.6594

roses

dandelion

tulips

sunflowers

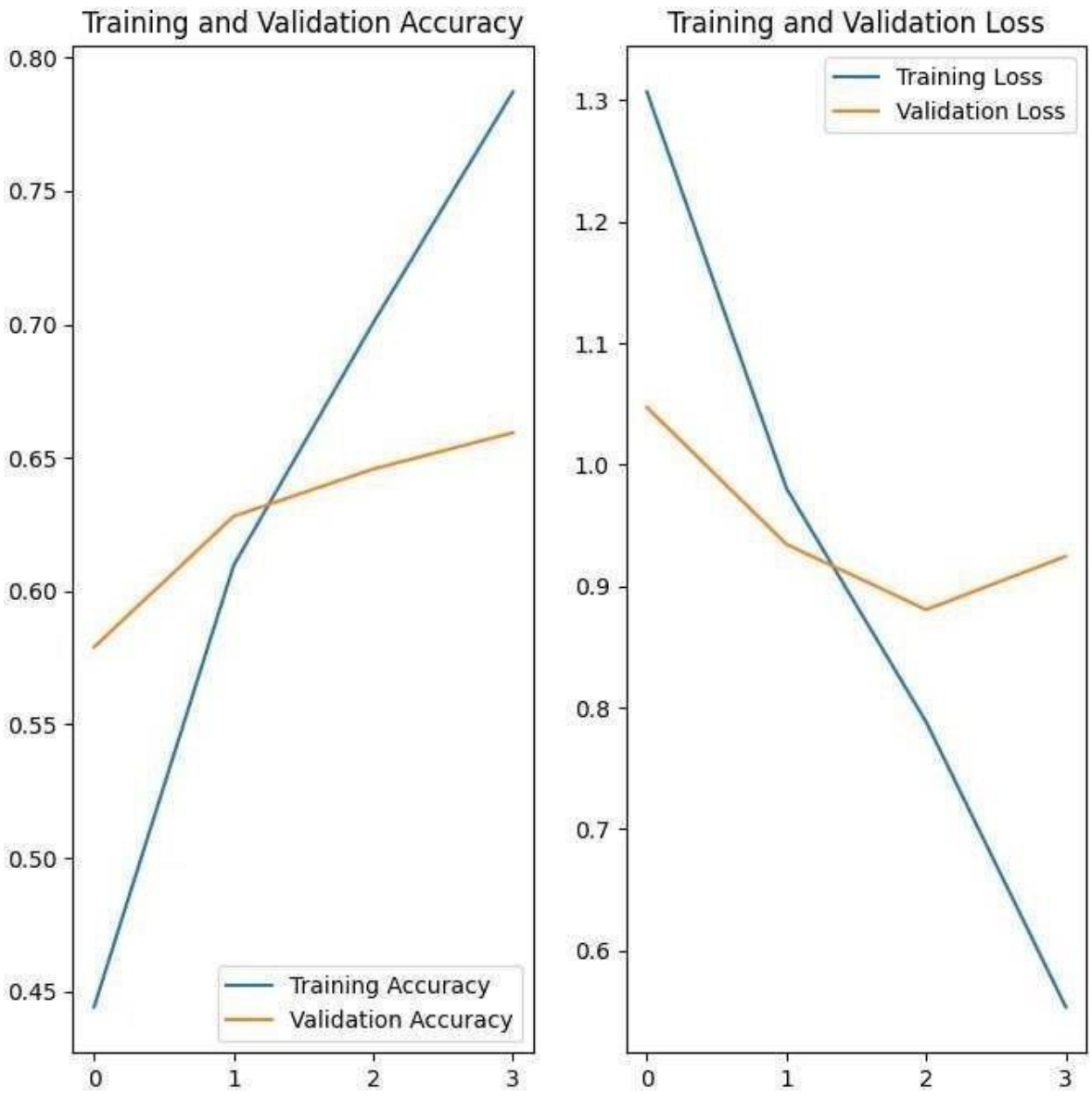
dandelion

roses

dandelion

roses

tulips



Result:

Thus, the program has been successfully executed for image classification using a deep learning model.

EX.NO:	Recommendation system from sales data using Deep Learning
DATE:	

Problem Statement

In the rapidly evolving landscape of e-commerce, personalized recommendations play a pivotal role in enhancing user experience and boosting sales. Our company, XYZ Retail, aims to leverage deep learning techniques to develop an advanced recommendation system based on historical sales data. The goal is to provide personalized product suggestions to users, ultimately increasing customer satisfaction and driving revenue.

Objective:

The objective of this project is to design and implement a deep learning-based recommendation system that can analyze past sales data and generate accurate product recommendations for individual users. The system should be capable of understanding user preferences, identifying patterns, and suggesting items that align with the user's interests.

Dataset used : sample_sales_data.csv

Code for generating the dataset using faker library to generate random names and words for users and items, respectively.

Step 1 : pip install faker Step 2

: Run the following code

import pandas as pd import

numpy as np from faker import

Faker import random import

datetime fake = Faker()

Generate sample users num_users

= 100

users = [fake.name() for _ in range(num_users)]

Generate sample items num_items

= 50

items = [fake.word() for _ in range(num_items)]

Generate sample sales data num_transactions

= 500

```
data = {
   'user': [random.choice(users) for _ in range(num_transactions)],
   'item': [random.choice(items) for _ in range(num_transactions)],
```

'purchase': [random.choice([0, 1]) for _ in range(num_transactions)],

'timestamp': [fake.date_time_between(start_date="-1y", end_date="now") for _ in range(num_transactions)]

}

sales_data = pd.DataFrame(data) #

Save the generated data to a CSV file

sales_data.to_csv('sample_sales_data.csv', index=False) #

Display the first few rows of the generated data

print(sales_data.head())

Code for Recommendation

System import pandas as pd import

numpy as np

from sklearn.model_selection import train_test_split from sklearn.preprocessing

import LabelEncoder from tensorflow.keras.models import Model from

tensorflow.keras.layers import Input, Embedding, Flatten, Dense, Concatenate

 # Load the sales data

data = pd.read_csv('/content/sample_sales_data.csv')

 # Preprocess data user_encoder

= LabelEncoder() item_encoder

= LabelEncoder()

data['user_id'] = user_encoder.fit_transform(data['user'])

data['item_id'] = item_encoder.fit_transform(data['item'])

 # Split data into training and testing sets

train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)

 # Define the model def create_model(num_users, num_items,

embedding_size=50): user_input = Input(shape=(1,),

name='user_input') item_input = Input(shape=(1,),

name='item_input')

user_embedding = Embedding(input_dim=num_users, output_dim=embedding_size, input_length=1)(user_input)

```
    item_embedding = Embedding(input_dim=num_items, output_dim=embedding_size,
input_length=1)(item_input)


    user_flatten = Flatten()(user_embedding)
item_flatten = Flatten()(item_embedding)     concat =
Concatenate()([user_flatten,  item_flatten])      dense1
= Dense(128, activation='relu')(concat)     dense2 =
Dense(64, activation='relu')(dense1)     output =
Dense(1, activation='sigmoid')(dense2)
    model = Model(inputs=[user_input, item_input], outputs=output)
    model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
return model
# Create and train the model num_users =
len(data['user_id'].unique()) num_items =
len(data['item_id'].unique()) model =
create_model(num_users, num_items)
model.summary()
train_user = train_data['user_id'].values
train_item = train_data['item_id'].values
train_labels = train_data['purchase'].values
model.fit([train_user, train_item], train_labels, epochs=5, batch_size=64,
validation_split=0.2)
# Evaluate the model
test_user = test_data['user_id'].values test_item =
test_data['item_id'].values test_labels =
test_data['purchase'].values accuracy =
model.evaluate([test_user, test_item], test_labels)
print(f'Test Accuracy: {accuracy[1]*100:.2f}%')
```

Output

```
Model: "model"
```

Layer (type)	Output Shape	Param #	Connected to
user_input (InputLayer)	[(None, 1)]	0	[]
item_input (InputLayer)	[(None, 1)]	0	[]
embedding (Embedding)	(None, 1, 50)	4950	['user_input[0][0]']
embedding_1 (Embedding)	(None, 1, 50)	2300	['item_input[0][0]']
flatten (Flatten)	(None, 50)	0	['embedding[0][0]']
flatten_1 (Flatten)	(None, 50)	0	['embedding_1[0][0]']
concatenate (Concatenate)	(None, 100)	0	['flatten[0][0]', 'flatten_1[0][0]']
dense (Dense)	(None, 128)	12928	['concatenate[0][0]']
dense_1 (Dense)	(None, 64)	8256	['dense[0][0]']
dense_2 (Dense)	(None, 1)	65	['dense_1[0][0]']

```
Total params: 28499 (111.32 KB)
Trainable params: 28499 (111.32 KB)
Non-trainable params: 0 (0.00 Byte)
```

```
Epoch 1/5
5/5 [==============================] - 2s 68ms/step - loss: 0.6937 - accuracy: 0.4938 - val_loss: 0.6948 - val_accuracy: 0.4125
Epoch 2/5
5/5 [==============================] - 0s 11ms/step - loss: 0.6864 - accuracy: 0.7281 - val_loss: 0.6955 - val_accuracy: 0.3875
Epoch 3/5
5/5 [==============================] - 0s 13ms/step - loss: 0.6791 - accuracy: 0.8000 - val_loss: 0.6970 - val_accuracy: 0.3625
Epoch 4/5
5/5 [==============================] - 0s 16ms/step - loss: 0.6696 - accuracy: 0.7937 - val_loss: 0.6993 - val_accuracy: 0.3750
Epoch 5/5
5/5 [==============================] - 0s 12ms/step - loss: 0.6555 - accuracy: 0.7875 - val_loss: 0.7039 - val_accuracy: 0.3750
4/4 [==============================] - 0s 6ms/step - loss: 0.6910 - accuracy: 0.4900
Test Accuracy: 49.00%
```

Result:

Thus, the program has been successfully executed for recommendation system from sales data using Deep Learning

EX.NO:	Implement Object Detection using CNN Problem
DATE:	

Statement:

In the realm of computer vision and image processing, object detection is a fundamental task that finds applications in various domains, including autonomous vehicles, surveillance, and augmented reality. Convolutional Neural Networks (CNNs) have proven to be powerful tools for object detection due to their ability to automatically learn hierarchical features from images.

Objective:

The objective of this project is to design and implement an Object Detection system using Convolutional Neural Networks. The system should be capable of accurately detecting and localizing objects within images, providing bounding box coordinates, class labels, and confidence scores.

Dataset Used: CIFAR-

10 dataset **Program**

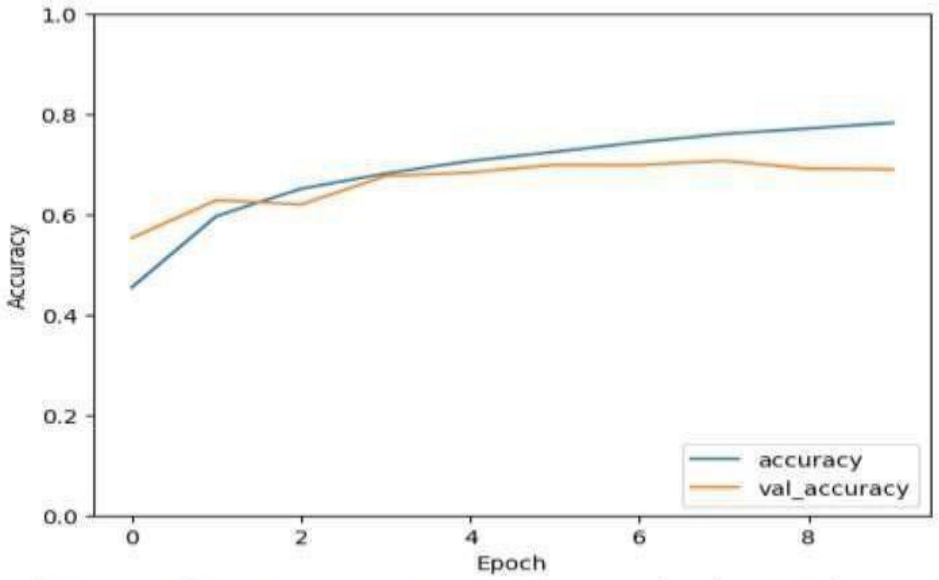
Code:

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import matplotlib.pyplot as plt
# Load CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()
# Normalize pixel values to be between 0 and 1
train_images, test_images = train_images / 255.0, test_images / 255.0
# Define the 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')
])
# Add dense layers for classification
model.add(layers.Flatten()) model.add(layers.Dense(64,
```

```
activation='relu')) model.add(layers.Dense(10)) # Compile the
model model.compile(optimizer='adam',
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])
# Train the model
history = model.fit(train_images, train_labels, epochs=10, validation_data=(test_images,
test_labels))


# Plot training history
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label =
'val_accuracy') plt.xlabel('Epoch') plt.ylabel('Accuracy')
plt.ylim([0, 1]) plt.legend(loc='lower right') plt.show()
# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels, verbose=2)
print('\nTest accuracy:', test_acc)
```

Output

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 11s 0us/step
Epoch 1/10
1563/1563 [==============================] - 82s 51ms/step - loss: 1.5050 - accuracy: 0.4550 - val_loss: 1.2289 - val_accuracy: 0.5537
Epoch 2/10
1563/1563 [==============================] - 67s 43ms/step - loss: 1.1410 - accuracy: 0.5972 - val_loss: 1.0598 - val_accuracy: 0.6288
Epoch 3/10
1563/1563 [==============================] - 65s 42ms/step - loss: 0.9956 - accuracy: 0.6515 - val_loss: 1.0731 - val_accuracy: 0.6203
Epoch 4/10
1563/1563 [==============================] - 67s 43ms/step - loss: 0.9049 - accuracy: 0.6821 - val_loss: 0.9332 - val_accuracy: 0.6769
Epoch 5/10
1563/1563 [==============================] - 68s 43ms/step - loss: 0.8305 - accuracy: 0.7069 - val_loss: 0.9185 - val_accuracy: 0.6838
Epoch 6/10
1563/1563 [==============================] - 67s 43ms/step - loss: 0.7771 - accuracy: 0.7252 - val_loss: 0.8801 - val_accuracy: 0.6994
Epoch 7/10
1563/1563 [==============================] - 66s 42ms/step - loss: 0.7266 - accuracy: 0.7445 - val_loss: 0.8842 - val_accuracy: 0.6991
Epoch 8/10
1563/1563 [==============================] - 68s 43ms/step - loss: 0.6827 - accuracy: 0.7608 - val_loss: 0.8775 - val_accuracy: 0.7075
Epoch 9/10
1563/1563 [==============================] - 65s 42ms/step - loss: 0.6483 - accuracy: 0.7719 - val_loss: 0.9336 - val_accuracy: 0.6917
Epoch 10/10
1563/1563 [==============================] - 65s 42ms/step - loss: 0.6123 - accuracy: 0.7831 - val_loss: 0.9455 - val_accuracy: 0.6902
```




```
313/313 - 4s - loss: 0.9455 - accuracy: 0.6902 - 4s/epoch - 13ms/step

Test accuracy: 0.6901999711990356
```

Result:

Thus, the program has been successfully executed for Object Detection using CNN

EX.NO:	**Implement any simple Reinforcement Algorithm for an NLP problem**
DATE:	

Problem Statement:

Design a text generation system that can autonomously generate coherent and meaningful text sequences based on a given initial prompt.

Aim:

The aim of this project is to implement a reinforcement learning-based approach, specifically Q-learning, to train a model to generate text sequences. The model should learn to select the most appropriate words to follow a given sequence of words, aiming to maximize the coherence and relevance of the generated text.

Algorithm:

1. **Import Libraries:** Import the necessary libraries including numpy, pandas, matplotlib, scikit-learn, and TensorFlow.
2. **Set Random Seeds:** Set random seeds to ensure reproducibility.
3. **Load Dataset:** Load the dataset containing Mastercard stock history from a CSV fileinto a pandas DataFrame. Drop irrelevant columns like "Dividends" and "Stock Splits". Print the first few rows and descriptive statistics of the dataset.
4. **Define Functions:**
   - **train_test_plot:** Plot the training and testing data split based on the specified start and end years.
   - **train_test_split:** Split the dataset into training and testing sets based onthe specified start and end years.
   - **split_sequence:** Define a function to split a univariate sequence into samples for time-series prediction.
   - **plot_predictions:** Plot the actual vs. predicted stock prices.
   - **return_rmse:** Calculate and print the root mean squared error (RMSE) between the actual and predicted stock prices.
5. **Preprocessing:**
   - Perform Min-Max scaling on the training set.
   - Split the training set into input features (X_train) and target variable (y_train) using the split_sequence function.
   - Reshape X_train for compatibility with LSTM input shape.
6. **Model Definition:**
   - Build an LSTM model using Keras Sequential API.

- Add an LSTM layer with 125 units and "tanh" activation function, followedby a Dense layer with 1 unit.
- Compile the model using RMSprop optimizer and mean squared error loss function.
- Print the summary of the model architecture.

7. **Model Training:**
   - Fit the LSTM model on the training data for 50 epochs with a batch size of32.
8. **Testing:**
   - Prepare the test data by taking the last n_steps (60) values from thedataset and scaling them.
   - Split the test data into input sequences (X_test) and corresponding targetvalues (y_test).
   - Reshape X_test for compatibility with the model's input shape.
   - Make predictions on the test data using the trained LSTM model.
   - Inverse transform the predicted values to obtain the actual stock prices.
9. **Evaluation:**
   - Plot the actual vs. predicted stock prices using the plot_predictions function.
   - Calculate and print the RMSE between the actual and predicted stockprices using the return_rmse function.

Program:

```
# Importing the libraries

import numpy as np import

pandas as pd import

matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean_squared_error

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.optimizers import SGD
```

```
from tensorflow.random import
set_seed set_seed(455)
np.random.seed(455)
dataset = pd.read_csv(
   "data/Mastercard_stock_history.csv", index_col="Date", parse_dates=["Date"]
).drop(["Dividends", "Stock Splits"], axis=1)
print(dataset.head())
print(dataset.describe())
dataset.isna().sum() tstart = 2016 tend =
2020 def train_test_plot(dataset, tstart,
tend):
   dataset.loc[f"{tstart}":f"{tend}", "High"].plot(figsize=(16, 4), legend=True)
dataset.loc[f"{tend+1}":, "High"].plot(figsize=(16, 4), legend=True)
plt.legend([f"Train (Before {tend+1})", f"Test ({tend+1} and beyond)"])
plt.title("MasterCard stock price")    plt.show()
train_test_plot(dataset,tstart,tend) def
train_test_split(dataset, tstart, tend):
   train = dataset.loc[f"{tstart}":f"{tend}", "High"].values
test = dataset.loc[f"{tend+1}":, "High"].values    return
train, test
training_set, test_set = train_test_split(dataset, tstart, tend)
sc = MinMaxScaler(feature_range=(0, 1)) training_set =
training_set.reshape(-1, 1) training_set_scaled =
sc.fit_transform(training_set)  def
split_sequence(sequence, n_steps):
   X, y = list(), list()    for i in
range(len(sequence)):
      end_ix = i + n_steps
      if end_ix > len(sequence) - 1:
         break
```

```
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
X.append(seq_x)
    y.append(seq_y)    return
np.array(X), np.array(y) n_steps
= 60 features = 1
# split into samples
X_train, y_train = split_sequence(training_set_scaled, n_steps)
# Reshaping X_train for model
X_train = X_train.reshape(X_train.shape[0],X_train.shape[1],features)
# The LSTM architecture model_lstm = Sequential() model_lstm.add(LSTM(units=125,
activation="tanh", input_shape=(n_steps, features))) model_lstm.add(Dense(units=1))
# Compiling the model
model_lstm.compile(optimizer="RMSprop", loss="mse")
model_lstm.summary()
model_lstm.fit(X_train, y_train, epochs=50, batch_size=32) dataset_total
= dataset.loc[:,"High"]
inputs = dataset_total[len(dataset_total) - len(test_set) - n_steps :].values inputs
= inputs.reshape(-1, 1)
#scaling
inputs = sc.transform(inputs)
# Split into samples
X_test, y_test = split_sequence(inputs, n_steps)
# reshape
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], features)
#prediction
predicted_stock_price = model_lstm.predict(X_test)
#inverse transform the values
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
def plot_predictions(test, predicted):
```

```
   plt.plot(test, color="gray", label="Real")
plt.plot(predicted, color="red", label="Predicted")
plt.title("MasterCard Stock Price Prediction")
plt.xlabel("Time")
   plt.ylabel("MasterCard Stock
Price")    plt.legend()    plt.show()

def return_rmse(test, predicted):
   rmse = np.sqrt(mean_squared_error(test, predicted))
print("The root mean squared error is {:.2f}.".format(rmse))
plot_predictions(test_set,predicted_stock_price)
return_rmse(test_set,predicted_stock_price)
```

Output:

```
Open      High      Low     Close     Volume
Date
2006-05-25  3.748967  4.283869  3.739664  4.279217  395343000
2006-05-26  4.307126  4.348058  4.103398  4.179680  103044000
2006-05-30  4.183400  4.184330  3.986184  4.093164   49898000 2006-05-31
4.125723  4.219679  4.125723  4.180608  30002000 2006-06-01  4.179678
4.474572  4.176887  4.419686   62344000


         Open        High         Low       Close       Volume
count  3872.000000  3872.000000  3872.000000  3872.000000  3.872000e+03
mean   104.896814   105.956054   103.769349   104.882714  1.232250e+07 std
106.245511  107.303589   105.050064   106.168693  1.759665e+07

min      3.748967     4.102467     3.739664     4.083861  6.411000e+05
25%      22.347203    22.637997    22.034458    22.300391  3.529475e+06
50%      70.810079    71.375896    70.224002    70.856083  5.891750e+06
75%     147.688448   148.645373   146.822013   147.688438  1.319775e+07
max     392.653890   400.521479   389.747812   394.685730  3.953430e+08
```

Open 0

High 0

Low 0

Close 0

Volume 0

dtype: int64

Model: "sequential"

_____

Layer (type) Output Shape Param #

==========

lstm (LSTM) (None, 125) 63500

_____

dense (Dense) (None, 1) 126

==========

Total params: 63,626

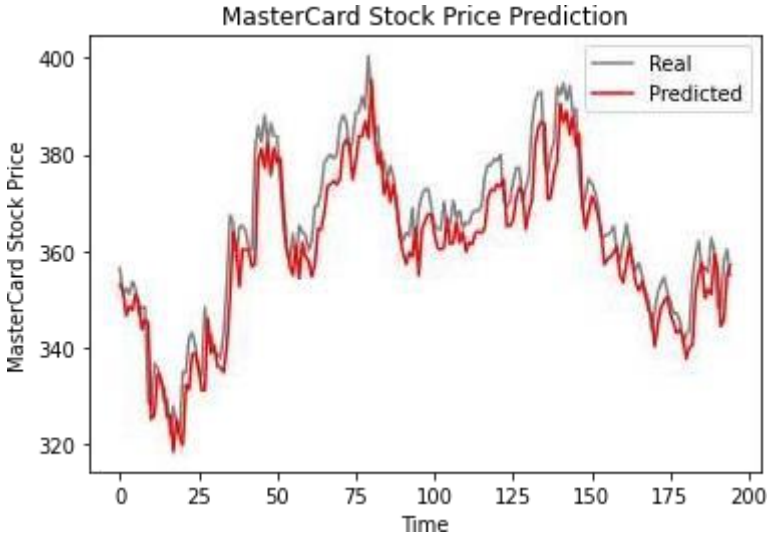
Trainable params: 63,626

Non-trainable params: 0

_____

Epoch 50/50

38/38 [==========] - 1s 30ms/step - loss: 3.1642e-04



>>> The root mean squared error is 6.70.

Result:

The program trains an LSTM model on Mastercard stock price data from 2016 to 2020 and tests its performance on data from 2021 onwards. It then plots the actual vs. predicted stock prices and calculates the root mean squared error (RMSE) between them. The root mean squared error is printed out, indicating the level of accuracy of the LSTM model in predicting the Mastercard stock prices.

CONTENT BEYOND SYLABBUS:

EX.NO:	**Implement a basic neural network for binary classification using tensor flow/keras.**
DATE:	

Problem Objective:

The objective of this problem is to develop a neural network model for binary classification.

Problem Statement:

Given a dataset with features and binary labels, the task is to design and train a neural network model that can accurately classify the samples into one of the two classes.

Aim:

To create a neural network model that effectively performs binary classification, achieving high accuracy on unseen data.

Algorithm:

STEP 1: Generate sample data for binary classification by randomly generating features and binary labels.

STEP 2: Create a sequential model with dense layers using ReLU activation functions and an output layer with a sigmoid activation function for binary classification.

STEP 3: Compile the model using Adam optimizer and binary cross-entropy loss function, with accuracy as the evaluation metric.

STEP 4: Train the model on the training data for a specified number of epochs and batch size, utilizing a portion for validation.

STEP 5: Evaluate the trained model's performance on the validation set, optionally on test data, and print the achieved accuracy.

STEP 6: Display training progress including loss and accuracy per epoch, optionally print accuracy on test data.

Program:

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

# Generate some sample data for binary classification
np.random.seed(42)
```

```
X_train = np.random.randn(1000, 10) # 1000 samples with 10 features
y_train = np.random.randint(2, size=1000) # Binary labels (0 or 1)

# Define the architecture of the neural
network model = models.Sequential([
layers.Dense(64, activation='relu',
input_shape=(10,)),          layers.Dense(32,
activation='relu'),
    layers.Dense(1, activation='sigmoid') # Output layer with sigmoid activation for binary classification
])

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

# Train the model model.fit(X_train, y_train, epochs=10,
batch_size=32, validation_split=0.2)

# Optionally, evaluate the model on test data
# X_test = np.random.randn(200, 10)  # Example test data
# y_test = np.random.randint(2, size=200) # Example test labels #
test_loss, test_acc = model.evaluate(X_test, y_test)
# print('Test accuracy:', test_acc)
```

Output:

```
Epoch 1/10
25/25 [==============================] - 1s 10ms/step - loss: 0.6963 - accuracy:
0.5400 - val_loss: 0.7098 - val_accuracy: 0.4050
Epoch 2/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6839 - accuracy: 0.5650 -
val_loss: 0.7086 - val_accuracy: 0.4350
Epoch 3/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6755 - accuracy: 0.5813 -
val_loss: 0.7070 - val_accuracy: 0.4650
Epoch 4/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6711 - accuracy: 0.5875 -
val_loss: 0.7085 - val_accuracy: 0.5000
Epoch 5/10
```

```
25/25 [==============================] - 0s 3ms/step - loss: 0.6670 - accuracy: 0.5813 - val_loss: 0.7074 - val_accuracy: 0.5100
Epoch 6/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6636 - accuracy: 0.6237 - val_loss: 0.7114 - val_accuracy: 0.4800
Epoch 7/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6608 - accuracy: 0.5950 - val_loss: 0.7083 - val_accuracy: 0.4950
Epoch 8/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6549 - accuracy: 0.6350 - val_loss: 0.7076 - val_accuracy: 0.4900
Epoch 9/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6512 - accuracy: 0.6488 - val_loss: 0.7097 - val_accuracy: 0.5000
Epoch 10/10
25/25 [==============================] - 0s 3ms/step - loss: 0.6479 - accuracy: 0.6425 - val_loss: 0.7100 - val_accuracy: 0.4800
<keras.src.callbacks.History at 0x7f888eadcb80>
```

Result:

After training the neural network model, it achieves a certain level of accuracy on the validation set, indicating its capability to classify binary data accurately. The accuracy obtained serves as a measure of the model's performance in classifying unseen data.