EXPERIMENT - 1

AIM: To explore the basic features of the TensorFlow and Keras packages in Python.

OBJECTIVES:

1. **Understanding the Fundamentals of Deep Learning Frameworks:** Familiarize yourself with TensorFlow and Keras, including their core concepts, APIs, and tools, to build and train machine learning models efficiently.

2. **Hands-on Implementation of Neural Networks:** Gain practical experience by implementing simple neural network models, understanding their architecture, and leveraging built-in functions for tasks such as data preprocessing, model creation, and evaluation.

THEORY:

TensorFlow is an open-source library developed by Google for numerical computation and large-scale machine learning, with a primary focus on deep learning models. It uses a dataflow graph to execute operations, allowing for computations to be scaled across multiple devices or distributed systems.

Keras is a high-level API that runs on top of TensorFlow. It's designed for fast and easy prototyping, offering intuitive tools for building and training neural networks.

CORE CONCEPTS

TensorFlow Basics

At its core, TensorFlow revolves around a few key ideas:

- **Tensors**: These are multi-dimensional arrays, similar to NumPy arrays, that serve as the fundamental data structure.

- **Graphs and Sessions**: TensorFlow represents computations as a graph of operations. Each node in the graph is a mathematical operation, and the edges represent the data (tensors) that flow between them. These operations are then executed within a session.

- **Eager Execution**: This feature allows for a more intuitive and flexible programming experience by evaluating operations immediately as they are called.

- **Optimizers**: These are algorithms, such as Gradient Descent and Adam, that adjust the model's parameters during training to minimize the loss function.

Keras Basics

Keras simplifies the process of building neural networks with the following components:

- **Models**: Keras provides two main ways to define a model: the **Sequential API** for simple, linear stacks of layers, and the **Functional API** for more complex architectures.
- **Layers**: These are the building blocks of a Keras model, such as Dense, Conv2D, and LSTM, each designed for different types of tasks.
- **Compilation**: Before training, a model must be compiled using the `model.compile()` method. This step configures the model with an optimizer, a loss function, and evaluation metrics.
- **Training and Evaluation**: The model is trained on data using the `model.fit()` method. After training, its performance can be assessed with `model.evaluate()` or used to make predictions on new data with `model.predict()`.

REGRESSION AND CLASSIFICATION

TensorFlow and Keras are well-suited for both regression and classification tasks.

Regression

Regression models predict continuous values, such as the price of a house or the temperature. In a typical Keras regression model:

- The output layer consists of a single neuron with no activation function, allowing for a linear output.
- Common loss functions include **Mean Squared Error (MSE)** or **Mean Absolute Error (MAE)**.
- The model is trained using an optimizer like **Adam** or **SGD**.
- Performance is often measured using metrics like MAE or Root Mean Squared Error (RMSE).

Classification

Classification models, on the other hand, predict discrete labels, such as whether an email is spam or not.

- **Binary Classification**: For tasks with two classes, the output layer has one neuron with a **sigmoid** activation function. The loss function used is typically **binary cross-entropy**.
- **Multi-Class Classification**: When there are more than two classes, the output layer uses a **softmax** activation function, with the number of neurons equal to the number of classes. The corresponding loss function is **categorical cross-entropy**.
- The model's performance is commonly evaluated using metrics like **accuracy** or the **F1-score**.

CODE & OUTPUT:

```
[1]  import pandas as pd
     import numpy as np
     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
```

```
[5]  data = pd.read_csv("housing.csv")

     X = data.drop("median_house_value", axis=1)
     y = data["median_house_value"]

     X = pd.get_dummies(X, columns=["ocean_proximity"], drop_first=True)

     scaler = StandardScaler()
     X_scaled = scaler.fit_transform(X)
```


```
[1]  import pandas as pd
     import numpy as np
     import tensorflow as tf
     from tensorflow import keras
     from tensorflow.keras import layers
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import StandardScaler
```

```
[5]  data = pd.read_csv("housing.csv")

     X = data.drop("median_house_value", axis=1)
     y = data["median_house_value"]

     X = pd.get_dummies(X, columns=["ocean_proximity"], drop_first=True)

     scaler = StandardScaler()
     X_scaled = scaler.fit_transform(X)
```


```
[6]  X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42)

     # regression model
     reg_model = keras.Sequential([
         layers.Dense(128, activation="relu", input_shape=(X_train.shape[1],)),
         layers.Dense(64, activation="relu"),
         layers.Dense(1)   # Linear output for regression
     ])

     reg_model.compile(optimizer="adam", loss="mse", metrics=["mae"])

     reg_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2, verbose=1)

     loss, mae = reg_model.evaluate(X_test, y_test, verbose=0)
     print(f"Regression Test MAE: {mae:.2f}")
```

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 3s 3ms/step - loss: 55682621440.0000 - mae: 206230.2344 - val_loss: 54635864064.0000 - val_mae: 203303.4062
Epoch 2/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 51928440832.0000 - mae: 197565.8750 - val_loss: 41150525440.0000 - val_mae: 173078.2188
Epoch 3/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 35821146112.0000 - mae: 159257.5000 - val_loss: 22211033088.0000 - val_mae: 119479.9766
Epoch 4/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 18603505664.0000 - mae: 108001.4141 - val_loss: 12300127232.0000 - val_mae: 83919.9062
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 18603505664.0000 - mae: 108001.4141 - val_loss: 12300127232.0000 - val_mae: 83919.9062
Epoch 5/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 10812191744.0000 - mae: 79362.9219 - val_loss: 9882185728.0000 - val_mae: 74542.3672
Epoch 6/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 9277438976.0000 - mae: 73494.6328 - val_loss: 8888481792.0000 - val_mae: 70059.9062
Epoch 7/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 8573791232.0000 - mae: 68965.2109 - val_loss: 8175582720.0000 - val_mae: 66636.2812
Epoch 8/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 7707225088.0000 - mae: 65287.5977 - val_loss: 7591735808.0000 - val_mae: 63803.4883
Epoch 9/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 7276620800.0000 - mae: 62346.8008 - val_loss: 7108146688.0000 - val_mae: 61285.2617
Epoch 10/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 6850398208.0000 - mae: 60372.5742 - val_loss: 6693421568.0000 - val_mae: 58905.2188
Epoch 11/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 6438217728.0000 - mae: 57935.1406 - val_loss: 6347557376.0000 - val_mae: 57160.2578
Epoch 12/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 6031987200.0000 - mae: 56040.9023 - val_loss: 6053023744.0000 - val_mae: 55520.1641
Epoch 13/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - loss: 5577364480.0000 - mae: 53946.1289 - val_loss: 5808775168.0000 - val_mae: 54200.1250
Epoch 14/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 5636777984.0000 - mae: 53390.9492 - val_loss: 5604462080.0000 - val_mae: 53269.9336
Epoch 15/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 5381288960.0000 - mae: 52809.6836 - val_loss: 5437854208.0000 - val_mae: 52320.6992
Epoch 16/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 5072574464.0000 - mae: 51410.3789 - val_loss: 5302561280.0000 - val_mae: 51764.7031
Epoch 17/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 5023481344.0000 - mae: 51039.0547 - val_loss: 5199497728.0000 - val_mae: 51200.6016
Epoch 18/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 4892037632.0000 - mae: 50162.6953 - val_loss: 5123300864.0000 - val_mae: 51084.4297
```

```
Epoch 18/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 4892037632.0000 - mae: 50162.6953 - val_loss: 5123300864.0000 - val_mae: 51084.4297
Epoch 19/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 4728552448.0000 - mae: 49272.5195 - val_loss: 5058136576.0000 - val_mae: 50774.2617
Epoch 20/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 4653617664.0000 - mae: 49159.7773 - val_loss: 4999168512.0000 - val_mae: 50373.0391
Regression Test MAE: nan
```

```python
# classification model
threshold = y.median()
y_class = (y >= threshold).astype(int)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_class, test_size=0.2, random_state=42)

clf_model = keras.Sequential([
    layers.Dense(128, activation="relu", input_shape=(X_train.shape[1],)),
    layers.Dense(64, activation="relu"),
    layers.Dense(1, activation="sigmoid")   # Sigmoid for binary classification
])

clf_model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])

clf_model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2, verbose=1)

loss, acc = clf_model.evaluate(X_test, y_test, verbose=0)
print(f"Classification Test Accuracy: {acc:.2f}")
```

```
Epoch 1/20
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.7959 - loss: 0.4463 - val_accuracy: 0.8435 - val_loss: 0.3459
Epoch 2/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8429 - loss: 0.3387 - val_accuracy: 0.8477 - val_loss: 0.3408
Epoch 3/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8564 - loss: 0.3211 - val_accuracy: 0.8526 - val_loss: 0.3300
Epoch 4/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8596 - loss: 0.3187 - val_accuracy: 0.8459 - val_loss: 0.3334
Epoch 5/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8676 - loss: 0.3091 - val_accuracy: 0.8526 - val_loss: 0.3265
Epoch 6/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8615 - loss: 0.3140 - val_accuracy: 0.8547 - val_loss: 0.3174
Epoch 7/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8703 - loss: 0.3030 - val_accuracy: 0.8526 - val_loss: 0.3124
Epoch 8/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8668 - loss: 0.3004 - val_accuracy: 0.8583 - val_loss: 0.3122
Epoch 9/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8716 - loss: 0.2967 - val_accuracy: 0.8565 - val_loss: 0.3112
Epoch 10/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8692 - loss: 0.2875 - val_accuracy: 0.8616 - val_loss: 0.3106
Epoch 11/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8707 - loss: 0.2887 - val_accuracy: 0.8607 - val_loss: 0.3012
Epoch 12/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8797 - loss: 0.2870 - val_accuracy: 0.8616 - val_loss: 0.3030
Epoch 13/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8763 - loss: 0.2805 - val_accuracy: 0.8644 - val_loss: 0.3011
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8849 - loss: 0.2698 - val_accuracy: 0.8568 - val_loss: 0.3029
Epoch 16/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8797 - loss: 0.2726 - val_accuracy: 0.8632 - val_loss: 0.2996
Epoch 17/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8833 - loss: 0.2735 - val_accuracy: 0.8486 - val_loss: 0.3213
Epoch 18/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8829 - loss: 0.2679 - val_accuracy: 0.8650 - val_loss: 0.2987
Epoch 19/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8805 - loss: 0.2700 - val_accuracy: 0.8616 - val_loss: 0.2950
Epoch 20/20
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - accuracy: 0.8814 - loss: 0.2719 - val_accuracy: 0.8650 - val_loss: 0.2956
Classification Test Accuracy: 0.86
```

CASE STUDIES

1. Predicting House Prices

Real estate companies need to accurately estimate property values based on various factors like location, size, and age. Traditional methods for this can be slow and may not capture complex market dynamics.

- **Application**: By leveraging TensorFlow and Keras, a regression model can be built using the Sequential API with `Dense` layers. This model can be trained on housing data, using features like longitude, latitude, and median income to predict the median house value.
- **Impact**: This enables real estate agents to provide competitive pricing, identify undervalued properties, and help clients make more informed decisions.

2. Medical Image Classification

In healthcare, doctors rely on medical imaging, such as X-rays and MRIs, to diagnose diseases. However, manual analysis can be time-consuming and prone to human error.

- **Application**: A classification model using Convolutional Neural Networks (CNNs) can be trained on large datasets of medical images. The model can be built with Keras layers like

  `Conv2D` and `MaxPooling`, with a final `Dense` layer using a **softmax** activation to classify images into categories like "normal" or "pneumonia.".

- **Impact**: This technology can significantly reduce diagnosis time, assist doctors in detecting diseases earlier, and improve patient outcomes, particularly in areas with limited resources.

3. Sentiment Analysis for Customer Feedback

Understanding customer sentiment is crucial for businesses to improve their products and services. Analyzing large volumes of text-based feedback, such as reviews and social media comments, can be a daunting task.

- **Application**: A classification model can be developed using Keras to perform sentiment analysis. By employing a Recurrent Neural Network (RNN) or a Long Short-Term Memory (LSTM) network, the model can be trained on a dataset of customer reviews labeled as positive, negative, or neutral. The model learns to recognize patterns and contextual nuances in the text to predict the sentiment of new, unseen feedback.

- **Impact**: This allows companies to automate the process of gauging customer opinion, identify areas for improvement, and respond to customer concerns more effectively. It can lead to enhanced customer satisfaction and better business outcomes.

LEARNING OUTCOME:

EXPERIMENT - 2

AIM: Implementation of Artificial Neural Network (ANN) model for regression and classification problem in Python.

OBJECTIVE:

1. To design and implement Artificial Neural Network (ANN) models using Python libraries for solving regression (predicting continuous values) and classification (categorical prediction) problems.

2. To evaluate the performance of ANN models using appropriate metrics.

THEORY:

Artificial Neural Network (ANN)

An Artificial Neural Network (ANN) is a computational model inspired by biological neural systems, consisting of layers of interconnected neurons that learn from data. ANNs are widely used for both **regression** (predicting continuous values) and **classification** (predicting categorical labels).

1. Structure of ANN

- **Input Layer:** Accepts dataset features.
- **Hidden Layers:** Perform weighted transformations using activation functions. ●

Output Layer: Produces final predictions (continuous or categorical).

2. Components of ANN

1. **Neuron (Perceptron):** Performs weighted sum and applies activation. 2.

Weights & Biases: Determine strength and flexibility of connections. 3.

Activation Functions:

- Regression: Linear, ReLU.
- Classification: Sigmoid (binary), Softmax (multi-class).

3. Learning Mechanism

- **Forward Propagation:** Inputs pass through layers to generate output.
- **Loss Function:** Measures error (MSE/MAE for regression, cross-entropy for classification).
- **Backpropagation:** Adjusts weights to minimize error.

- **Optimizers:** Gradient Descent, Adam, and RMSProp update weights efficiently.

CORE CONCEPTS:

Artificial neural networks are built on several fundamental concepts that enable them to learn and generalize from data for regression and classification tasks.

1. Neuron (Perceptron)

- Basic computational unit of ANN.
- Performs a weighted sum of inputs and applies an activation function. ●

Equation:

$y=f(\sum w_i x_i + b)$ y = f(\sum w_i x_i + b)y=f(∑wixi+b)

2. Network Architecture

1. Input Layer: Accepts raw features.
2. Hidden Layers: Extract higher-level patterns.
3. Output Layer: Produces prediction (continuous or categorical). 4.

Types of Architectures:

- Shallow ANN: Few hidden layers.
- Deep ANN: Multiple hidden layers (Deep Learning).

3. Activation Functions

- Introduce non-linearity, allowing the model to capture complex patterns. ●

Common Types:

- Sigmoid: Maps output between 0 and 1 (binary classification).
- ReLU: Efficient and avoids vanishing gradients (regression & hidden layers). ○ Softmax: Converts outputs to probabilities for multi-class classification.
- Linear: Direct mapping for regression outputs.

4. Loss Functions

- Measure error between predicted and actual values.
- Regression: Mean Squared Error (MSE), Mean Absolute error (MAE).
- Classification: Binary Cross-Entropy, Categorical Cross-Entropy.

5. Training Process

1. Forward Propagation: Data flows through layers to produce output.
2. Backpropagation: Computes gradients of loss w.r.t. weights.
3. Optimization: Updates weights to minimize loss (Gradient Descent, Adam).

Regression and Classification using ANN

ANN for Regression

- Purpose: Predicts continuous outputs (e.g., house price, temperature).
- Output Layer: Linear/ReLU activation.
- Evaluation Metrics: Mean Squared Error (MSE), Mean Absolute Error (MAE), R^2 Score.

ANN for Classification

- Purpose: Categorizes data into classes (e.g., spam detection, disease prediction).
- Output Layer: Sigmoid for binary, Softmax for multi-class.
- Evaluation Metrics: Accuracy, Precision, Recall, F1-Score.

CODE & OUTPUT:

```
import pandas as pd
import tensorflow as tf
housing = tf.keras.datasets.boston_housing
(x_train, y_train), (x_test, y_test) = housing.load_data()
model = tf.keras.models.Sequential([
tf.keras.layers.Dense(64, activation='relu',
input_shape=(x_train.shape[1],)),
tf.keras.layers.Dense(64, activation='relu'),
tf.keras.layers.Dense(1)
])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/boston_housing.npz
57026/57026 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
[ ] model.compile(optimizer='sgd', loss='mse', metrics=['mae'])
    history = model.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.2)


    test_loss, test_mae = model.evaluate(x_test, y_test)
    print(f"Test MAE: {test_mae}")

    predictions = model.predict(x_test)
```

```
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 74/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 75/100
```

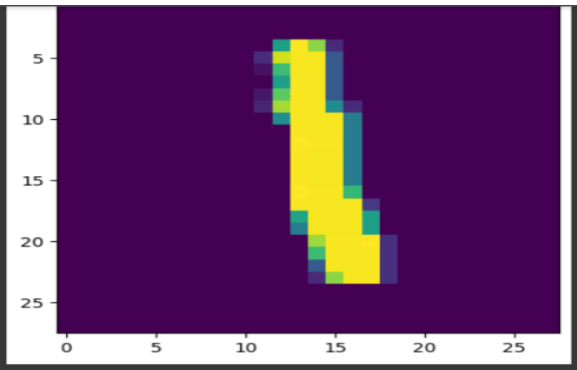
```
Epoch 92/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 93/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 94/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 95/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 96/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 97/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 98/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 99/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
Epoch 100/100
11/11 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan - val_loss: nan - val_mae: nan
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: nan - mae: nan
Test MAE: nan
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step
```

```
import tensorflow as tf
from keras.layers import Dense, Flatten, Input
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

```
((60000, 28, 28), (60000,), (10000, 28, 28), (10000,))
```

```
import matplotlib.pyplot as plt
plt.imshow(x_train[6])
model = tf.keras.models.Sequential([
    Input(shape=(28,28)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size = 100, verbose =1)
loss,acc = model.evaluate(x_test, y_test)
loss,acc
```

```
Epoch 1/5
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.7930 - loss: 6.5896
Epoch 2/5
600/600 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9286 - loss: 0.4640
Epoch 3/5
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9485 - loss: 0.2455
Epoch 4/5
600/600 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.9584 - loss: 0.1878
```




```
import numpy as np
pred = model.predict(x_train[6].reshape(1,28,28))
np.argmax(pred)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 68ms/step
np.int64(1)
```

CASE STUDY:

Case Study 1: Retail Demand Forecasting (Regression using ANN)

- **Problem:** Stockouts and overstocking hurt revenue and cash flow.
- **ANN Application:**
    - **Input Layer**: Historical sales, promotions, price, holidays, competitor price index, weather, store demographics.
    - **Hidden Layers**: Learn promo-lift curves, price elasticity, holiday effects across SKUs/stores.
    - **Output Layer:** Next-period demand (units) per SKU-store.
- **Outcome:** Smarter replenishment, lower inventory costs, higher on-shelf availability.

Case Study 2: Credit Card Fraud Detection (Classification using ANN)

- **Problem:** Real-time detection of fraudulent transactions with highly imbalanced data.
- **ANN Application:**
    - **Input Layer**: Transaction amount, merchant type, geo, device ID, customer behavior profiles, time gaps.
    - **Hidden Layers:** Capture subtle interactions (amount×merchant×time) and user-specific behavior drifts.
    - **Output Layer**: Fraud probability → “Fraud” / “Legit.”
- **Outcome:** Fewer false declines, faster blocking of fraud, reduced financial loss.

LEARNING OUTCOME:

EXPERIMENT - 3

AIM: Implementation of Convolution Neural Network for MRI Data Set in Python.

OBJECTIVE:

1. Implement a CNN model using TensorFlow/Keras to classify MRI images (e.g., tumor vs. normal).
2. Optimize model architecture, activation functions, and hyperparameters for improved accuracy.

THEORY:

Convolutional Neural Network (CNN)

A **Convolutional Neural Network (CNN)** is a deep learning model designed to process structured grid data such as images. CNNs are inspired by the visual cortex and are highly effective in extracting spatial features from images. They are widely used in computer vision tasks like classification, detection, and segmentation.

1. Structure of CNN

- **Input Layer:** Accepts MRI image data (2D/3D pixel intensity values).
- **Convolutional Layers:** Apply filters/kernels to extract local features (edges, textures, patterns).
- **Activation Functions:** Non-linear transformations (ReLU, Sigmoid, Softmax).
- **Pooling Layers:** Downsample feature maps to reduce dimensionality (Max Pooling, Average Pooling).
- **Fully Connected Layers:** Combine extracted features for final prediction.
- **Output Layer:** Produces prediction (disease/no disease, tumor type, etc.).

2. Components of ANN

1. **Convolution Operation**
    - Applies kernels/filters across the input image to detect spatial features**.**
2. **Filters/Kernels**
    - Learn feature detectors (edges, textures, shapes).
    - Early layers learn simple features, deeper layers learn complex features.
3. **Pooling Layers**
    - Reduce spatial dimensions and computational cost.
    - Max pooling selects strongest activation in a region.
4. **Flattening & Fully Connected Layers**
    - Convert 2D feature maps into 1D vectors.
    - Feed into dense layers for classification.

5. **Activation Functions**
    - **ReLU:** Prevents vanishing gradients, adds non-linearity.
    - **Softmax:** Converts logits into class probabilities (multi-class MRI classification).
    - **Sigmoid:** For binary medical classification (e.g., tumor vs. no tumor).

3. Why CNN for MRI Dataset?

- MRI images are high-dimensional medical images where critical patterns like tumor boundaries may not be visible to the human eye easily.

- CNNs are capable of automatically detecting these subtle patterns.

- They significantly improve diagnostic accuracy compared to traditional machine learning models, which rely on handcrafted features.

4. Workflow of CNN on MRI Dataset

1. Data Preprocessing
    - Resize MRI scans to a fixed dimension.
    - Normalize pixel values (0–1).
    - Augment data (rotation, flipping) to improve generalization.

2. Model Training
    - Feed MRI images into CNN layers.
    - Backpropagation updates weights using loss function (e.g., binary cross-entropy).

3. Model Evaluation
    - Metrics: Accuracy, Precision, Recall, F1-score.
    - Used to check how well the CNN distinguishes between healthy and abnormal MRI scans.

CORE CONCEPTS:

1. Convolution Operation

- Fundamental step in CNNs where small filters (kernels) scan across MRI images.
- Captures local spatial features such as edges, shapes, and textures.

2. Feature Maps

- Output of convolution layers that highlight important regions of MRI scans.
- Represent hierarchical features: low-level (edges) → high-level (tumor patterns).

3. Pooling (Subsampling/Downsampling)

- Reduces spatial size of feature maps using Max/Average pooling.
- Preserves essential features while reducing computation and overfitting.

4. Activation Functions

- Introduce non-linearity into the network.
- ReLU (Rectified Linear Unit) is most common in CNNs for faster convergence.

5. Fully Connected Layers (Dense Layers)

- Convert extracted spatial features into final classification.
- Helps map learned features to specific MRI categories (e.g., normal vs tumor).

6. Backpropagation & Optimization

- Loss function (e.g., Binary Cross-Entropy) computes prediction error.
- Optimizers like Adam/SGD update weights to minimize loss.

7. Regularization Techniques

- Dropout: Randomly deactivates neurons to prevent overfitting
- Batch Normalization: Stabilizes learning and speeds up convergence.

8. Evaluation Metrics

- Accuracy: Overall correctness.
- Precision & Recall: Important in medical imaging where false negatives (missed tumors) must be minimized.

F1-Score: Balanced metric for performance evaluation

CODE & OUTPUT:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Input, Conv2D, MaxPooling2D, Flatten, Dense,
                                     Dropout, BatchNormalization)
import os
import kagglehub
import pandas as pd
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.model_selection import train_test_split
```

```
kagglehub.dataset_download("navoneel/brain-mri-images-for-brain-tumor-detection")
```

```
data_path = "/kaggle/input/brain-mri-images-for-brain-tumor-detection"
images = []
labels = []

for label_name in os.listdir(data_path):
    label_dir = os.path.join(data_path, label_name)
    if os.path.isdir(label_dir):
        for img_file in os.listdir(label_dir):
            images.append(os.path.join(label_dir, img_file))
            labels.append(label_name)

data_df = pd.DataFrame({
    "Image_Path": images,
    "Label": labels
})
```

```
train, test = train_test_split(data_df, test_size=0.2, stratify=data_df['Label'], random_state=42)
```

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=20,
    zoom_range=0.15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)

test_datagen = ImageDataGenerator(rescale=1./255)

train_gen = train_datagen.flow_from_dataframe(
    train,
    x_col='Image_Path',
    y_col='Label',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    shuffle=True
)

test_gen = test_datagen.flow_from_dataframe(
    test,
    x_col='Image_Path',
    y_col='Label',
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    shuffle=False
)
```

```
[9] model = Sequential([
        Input(shape=(224,224,3)),

        Conv2D(32, (3,3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2,2)),

        Conv2D(64, (3,3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2,2)),

        Conv2D(128, (3,3), activation='relu'),
        BatchNormalization(),
        MaxPooling2D(pool_size=(2,2)),

        Flatten(),
        Dense(128, activation='relu'),
        Dropout(0.5),
        Dense(2, activation='softmax')
    ])
```

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

```
model.fit(
    train_gen,
    validation_data=(test_gen),
    epochs=10
)
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class should cal
  self._warn_if_super_not_called()
Epoch 1/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 24s 2s/step - accuracy: 0.5931 - loss: 9.1513 - val_accuracy: 0.6471 - val_loss: 0.7854
Epoch 2/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 387ms/step - accuracy: 0.7308 - loss: 11.6314 - val_accuracy: 0.4510 - val_loss: 4.1169
Epoch 3/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 393ms/step - accuracy: 0.6222 - loss: 8.0951 - val_accuracy: 0.3922 - val_loss: 9.3493
Epoch 4/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 452ms/step - accuracy: 0.6731 - loss: 3.1102 - val_accuracy: 0.4118 - val_loss: 6.3076
Epoch 5/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 4s 522ms/step - accuracy: 0.6043 - loss: 3.0932 - val_accuracy: 0.5490 - val_loss: 2.4198
Epoch 6/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 381ms/step - accuracy: 0.7551 - loss: 1.3989 - val_accuracy: 0.6863 - val_loss: 2.1153
Epoch 7/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 463ms/step - accuracy: 0.6861 - loss: 0.7448 - val_accuracy: 0.6275 - val_loss: 5.2933
Epoch 8/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 386ms/step - accuracy: 0.6633 - loss: 0.6648 - val_accuracy: 0.6078 - val_loss: 7.5658
Epoch 9/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 4s 521ms/step - accuracy: 0.6761 - loss: 0.5420 - val_accuracy: 0.6078 - val_loss: 8.1461
Epoch 10/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 3s 383ms/step - accuracy: 0.6871 - loss: 0.6072 - val_accuracy: 0.6078 - val_loss: 9.8695
<keras.src.callbacks.history.History at 0x7dda093ba090>
```

CASE STUDY:

Case Study 1: Brain Tumor Detection Using CNN

- Purpose: Identify disease type (e.g., tumor detection/classification).
- Input: Preprocessed MRI scans (resized, normalized).
- Architecture: Convolution → ReLU → Pooling → Dropout → Dense → Softmax.

Output Layer:

- Sigmoid (binary classification: tumor/no tumor).

  - Softmax (multi-class classification: tumor types).

- Evaluation: Accuracy, Precision, Recall, F1-Score, Confusion Matrix.

Case Study 2: Spinal Cord Injury Classification using CNN

- **Objective:** Detect and classify spinal cord injuries from MRI scans for faster clinical assessment.

- **Method:** A CNN model was trained on a curated MRI dataset containing healthy and injured spinal cord images.

- Process:
  - MRI scans were preprocessed by resizing, intensity normalization, and removing noise.
  - A CNN with multiple convolution + max pooling layers was designed to capture spinal cord structure.
  - Dropout was added to prevent overfitting, and training used the Adam optimizer with categorical cross-entropy loss.

- Outcome: Achieved ~93% classification accuracy in differentiating healthy vs. injured spinal cord regions.

- Impact: Provided radiologists with an AI-based decision support system, reducing diagnosis time and improving injury management planning.

LEARNING OUTCOME:

EXPERIMENT 4

AIM: Implementation of autoencoders for dimensionality reduction in Python.

OBJECTIVES:

1. To design and train an autoencoder model in Python that learns compact, lower-dimensional representations of high-dimensional input data while minimizing reconstruction error.

2. To apply the learned low-dimensional features for tasks such as data visualization or preprocessing, thereby demonstrating the effectiveness of autoencoders in dimensionality reduction.

THEORY:

1. Introduction

Dimensionality reduction is the process of reducing the number of input variables in a dataset while preserving essential information. It is widely used in data visualization, noise removal, and improving computational efficiency. Traditional methods such as Principal Component Analysis (PCA) work well for linear data, but they often fail to capture nonlinear relationships. To overcome this limitation, autoencoders—a type of neural network—are used for nonlinear dimensionality reduction.

2. What is an Autoencoder?

An autoencoder is an unsupervised learning model that attempts to reconstruct its input at the output. It consists of two main components:

- Encoder: Compresses input data into a smaller representation (latent space).
- Decoder: Reconstructs the original data from the compressed representation.

The network is trained to minimize reconstruction error, ensuring that the latent space captures the most important features of the data.

3. Working Principle

- Input data is passed through the encoder, which applies transformations to reduce dimensions.
- The compressed latent vector (bottleneck layer) holds the most relevant features.
- The decoder reconstructs the data from this compressed representation.

- Training continues until the output closely resembles the input, using a loss function such as Mean Squared Error (MSE).

4. Advantages over Traditional Methods

- Captures nonlinear relationships unlike PCA.
- Learns data-driven representations automatically.
- Can handle large and complex datasets.
- Extensions like Denoising Autoencoders (noise removal) and Variational Autoencoders (probabilistic modeling) provide additional flexibility.

5. Applications in Dimensionality Reduction

- Data Visualization: Mapping high-dimensional data to 2D or 3D space.
- Preprocessing: Providing compact input for machine learning models.
- Clustering & Classification: Using latent features for improved accuracy.
- Noise Reduction: Extracting clean representations from noisy data.

CORE CONCEPTS:

1. Encoder–Decoder Architecture

The autoencoder is based on two parts:

- Encoder: Transforms input data into a compact latent vector by applying layers of neurons.
- Decoder: Reconstructs the original input from the latent vector, ensuring minimal information loss.

2. Latent Space (Bottleneck Layer)

The middle layer, called the bottleneck, holds the compressed version of input data. This acts as the reduced-dimensional representation, useful for visualization or further analysis.

3. Reconstruction Loss

The learning objective of autoencoders is to minimize the difference between the input and output.

- Common metrics: Mean Squared Error (MSE) or Cross-Entropy Loss.
- Lower loss indicates better dimensionality reduction with minimal distortion.

4. Nonlinear Feature Learning

Unlike PCA, which captures only linear correlations, autoencoders can extract nonlinear and complex features from data, making them effective for images, text, and high-dimensional datasets.

5. Variants of Autoencoders

- Denoising Autoencoder: Learns robust features by reconstructing inputs from noisy versions.
- Variational Autoencoder (VAE): Learns probabilistic latent representations, enabling generative modeling.

CODE & OUTPUT:

```python
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import matplotlib.pyplot as plt

iris = load_iris()
X = iris.data
y = iris.target


scaler = StandardScaler()
X_normalized = scaler.fit_transform(X)

X_train, X_test = train_test_split(X_normalized, test_size=0.2, random_state=42)

input_data = Input(shape=(4,))

encoded = Dense(3, activation='relu')(input_data)
encoded = Dense(2, activation='relu')(encoded)
decoded = Dense(3, activation='relu')(encoded)
decoded = Dense(4, activation='sigmoid')(decoded)
autoencoder = Model(input_data, decoded)

encoder = Model(input_data, encoded)
```

```python
encoder = Model(input_data, encoded)

autoencoder.compile(optimizer='adam', loss='mean_squared_error', metrics = ["accuracy"])

autoencoder.fit(X_train, X_train, epochs=100, batch_size=16, shuffle=True, validation_data=(X_test, X_test))

X_encoded = encoder.predict(X_test)

print("Encoded shape:", X_encoded.shape)
```

```
Epoch 73/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.3497 - loss: 0.8174 - val_accuracy: 0.4667 - val_loss: 0.7487
Epoch 74/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.3498 - loss: 0.7932 - val_accuracy: 0.4667 - val_loss: 0.7479
Epoch 75/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.3350 - loss: 0.8413 - val_accuracy: 0.4667 - val_loss: 0.7471
Epoch 76/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.3641 - loss: 0.8440 - val_accuracy: 0.4667 - val_loss: 0.7462
Epoch 77/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.2657 - loss: 0.8940 - val_accuracy: 0.4667 - val_loss: 0.7454
Epoch 78/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.3144 - loss: 0.8337 - val_accuracy: 0.4667 - val_loss: 0.7446
Epoch 79/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.3184 - loss: 0.8302 - val_accuracy: 0.4667 - val_loss: 0.7438
```

```
Epoch 97/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.3278 - loss: 0.7478 - val_accuracy: 0.4667 - val_loss: 0.7310
Epoch 98/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 18ms/step - accuracy: 0.2785 - loss: 0.8630 - val_accuracy: 0.4667 - val_loss: 0.7304
Epoch 99/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.3263 - loss: 0.8191 - val_accuracy: 0.4667 - val_loss: 0.7298
Epoch 100/100
8/8 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.3110 - loss: 0.8029 - val_accuracy: 0.4667 - val_loss: 0.7292
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 94ms/step
Encoded shape: (30, 2)
```

CASE STUDY:

Case Study 1: Handwritten Digit Recognition (MNIST Dataset)

The MNIST dataset of handwritten digits (0–9) contains 28×28 pixel grayscale images. Directly working with 784-dimensional input data can be computationally expensive. An autoencoder was applied to compress the data

into a 32-dimensional latent space, preserving essential features while discarding noise.
The compressed features were then used for visualization and classification. Results showed that the autoencoder effectively reduced dimensionality while maintaining accuracy, outperforming PCA for digit recognition tasks.
Case Study 2: Genomic Data Compression in Bioinformatics

Genomic datasets often contain tens of thousands of features representing gene expressions. Traditional methods fail to capture nonlinear patterns in such data. Researchers used autoencoders to reduce the dimensionality of RNA sequencing data into a compact latent space. This reduced representation was then applied to cancer classification, helping identify patterns and clusters in patients. Autoencoders not only reduced computational cost but also improved prediction accuracy in comparison to standard reduction methods.

LEARNING OUTCOME:

09717711623 Ujjwal Chaurasia AI&ML - B

EXPERIMENT 5

AIM: Application of Autoencoders on Image Dataset.

OBJECTIVES:

1. To explore the application of autoencoders on an image dataset
2. The aim is to train to autoencoder to compress and reconstruct images, thereby enabling tasks such as image compression, feature extraction, and noise reduction

THEORY:

1. Introduction

In image processing, datasets often contain thousands of pixels per image, leading to highdimensional data. This creates challenges in computation, storage, and pattern extraction. Autoencoders, a deep learning model, are widely used for feature extraction, noise removal, and image compression.

2. Structure of an Autoencoder

An autoencoder consists of two main components:

- Encoder: Maps the high-dimensional image into a lower-dimensional latent space. ● Decoder: Reconstructs the original image from this compressed representation.

The network is trained to minimize the reconstruction loss, ensuring the latent features capture the essential characteristics of the image.

3. Working Principle

1. Input image → Encoder → Latent vector (compressed representation).
2. Latent vector → Decoder → Reconstructed image.
3. The difference between input and output (loss) is minimized during training.

4. Benefits in Image Data

- Dimensionality Reduction: Images can be compressed without losing critical details.
- Noise Removal: Denoising autoencoders can reconstruct clean images from corrupted ones.
- Feature Learning: Extracted latent features can be used for clustering, classification, and visualization.

CORE CONCEPTS:

Encoder–Decoder Architecture

- Encoder compresses the image into a smaller latent space.
- Decoder reconstructs the image from this compressed data.

Latent Space Representation

- The bottleneck layer holds the compressed features of images. ● Useful for visualization, clustering, or as input to classifiers.

Reconstruction Loss

- Measures the difference between original and reconstructed images.
- Common metrics: Mean Squared Error (MSE), Binary Cross-Entropy.

Nonlinear Feature Learning

- Captures complex patterns in image data beyond linear methods like PCA.

Variants for Images

- Denoising Autoencoders → Remove noise from corrupted images.
- Convolutional Autoencoders → Use convolutional layers to learn spatial features of images.

CODE & OUTPUT:

```
[ ] import numpy as np
    import matplotlib.pyplot as plt
    import tensorflow as tf
    from tensorflow import keras
    from tensorflow.keras import layers, losses
    from tensorflow.keras.models import Model
    from keras.datasets import mnist
```

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

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

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

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

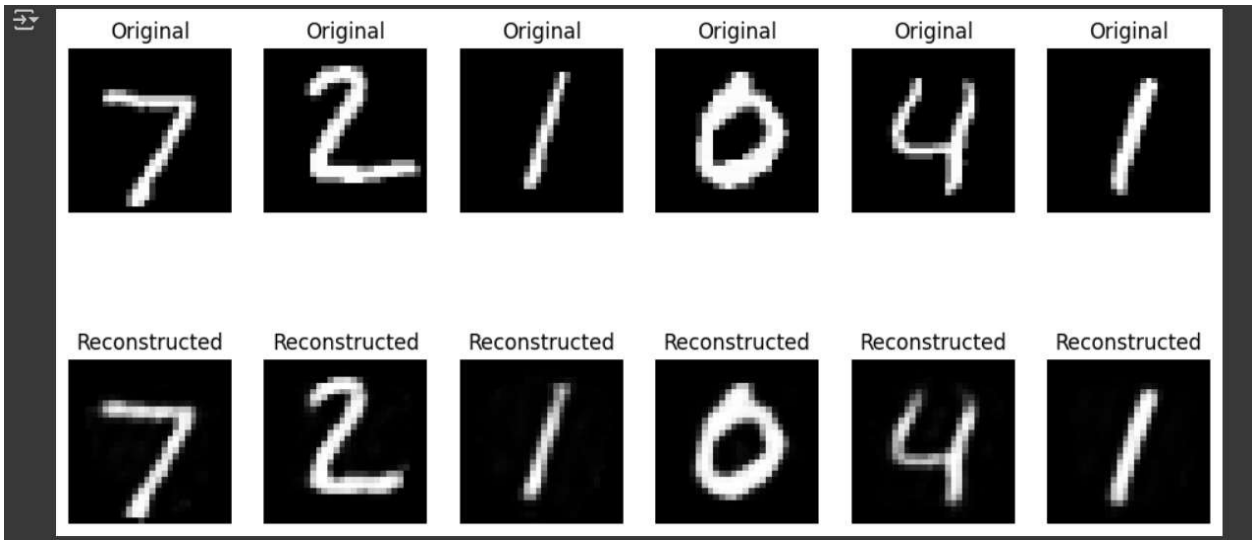
```
[ ] class SimpleAutoencoder(Model):
        def __init__(self, latent_dimensions):
            super(SimpleAutoencoder, self).__init__()
            self.encoder = tf.keras.Sequential([
                layers.Input(shape=(28, 28, 1)),
                layers.Flatten(),
                layers.Dense(latent_dimensions, activation='relu'),
            ])

            self.decoder = tf.keras.Sequential([
                layers.Dense(28 * 28, activation='sigmoid'),
                layers.Reshape((28, 28, 1))
            ])

        def call(self, input_data):
            encoded = self.encoder(input_data)
            decoded = self.decoder(encoded)
            return decoded
```

```
latent_dimensions = 64
autoencoder = SimpleAutoencoder(latent_dimensions)
autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError(), metrics = ["accuracy"])

autoencoder.fit(x_train, x_train,
                epochs=10,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.7528 - loss: 0.0952 - val_accuracy: 0.8043 - val_loss: 0.0334
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - accuracy: 0.8063 - loss: 0.0302 - val_accuracy: 0.8098 - val_loss: 0.0211
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - accuracy: 0.8113 - loss: 0.0195 - val_accuracy: 0.8119 - val_loss: 0.0146
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - accuracy: 0.8132 - loss: 0.0140 - val_accuracy: 0.8130 - val_loss: 0.0111
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8141 - loss: 0.0109 - val_accuracy: 0.8136 - val_loss: 0.0090
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8143 - loss: 0.0088 - val_accuracy: 0.8139 - val_loss: 0.0075
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - accuracy: 0.8152 - loss: 0.0075 - val_accuracy: 0.8140 - val_loss: 0.0065
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - accuracy: 0.8149 - loss: 0.0066 - val_accuracy: 0.8142 - val_loss: 0.0059
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 15ms/step - accuracy: 0.8152 - loss: 0.0059 - val_accuracy: 0.8142 - val_loss: 0.0054
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.8152 - loss: 0.0055 - val_accuracy: 0.8142 - val_loss: 0.0051
<keras.src.callbacks.history.History at 0x796e16468560>
```

+ Code + Text

```python
encoded_imgs = autoencoder.encoder(x_test).numpy()
decoded_imgs = autoencoder.decoder(encoded_imgs).numpy()

n = 6
plt.figure(figsize=(12, 6))
for i in range(n):
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')

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

plt.show()
```



CASE STUDY:

Case Study 1: Image Denoising using Autoencoders

Researchers applied denoising autoencoders on the CIFAR-10 dataset by intentionally corrupting images with Gaussian noise. The autoencoder successfully reconstructed clean images, proving effective in noise reduction for real-world image datasets such as medical imaging and photography.

Case Study 2: Image Compression on Fashion-MNIST Dataset

The Fashion-MNIST dataset, containing grayscale clothing images, was used to test autoencoders for compression. Images with 784 dimensions (28×28 pixels) were reduced to a latent space of 64 dimensions. The compressed features were then used to reconstruct the original images with minimal loss. This reduced storage needs while preserving visual clarity, making it useful for real-time image transmission and storage.

LEARNING OUTCOME:

EXPERIMENT - 6

AIM: Improving Autoencoder's Performance using convolution layers in Python.

OBJECTIVE:

The objective of this experiment is to improve the performance of traditional autoencoders by incorporating convolutional layers, using the MNIST dataset as a case study. While basic autoencoders use fully connected layers, they may fail to effectively capture the spatial hierarchies in image data. Convolutional autoencoders, on the other hand, exploit the spatial structure of images through shared weights and local connectivity, leading to better feature extraction and more accurate image reconstruction.

THEORY:

Traditional autoencoders rely on fully connected (dense) layers which flatten the 2D image data into a 1D vector. This operation leads to the loss of important spatial information such as edges, local neighborhoods, and positional dependencies. As a result, reconstructions often appear blurry and fine structural details like digit curves or intersections are lost.

Convolutional Autoencoders (CAEs) resolve this limitation by preserving the two-dimensional structure of input images. By applying convolutional filters, they exploit three key advantages:

- **Local connectivity:** Each neuron connects only to a small region of the input, improving the capture of localized patterns (e.g., a horizontal line in the digit "7").
- **Weight sharing:** Filters are reused across the entire image, greatly reducing the number of learnable parameters and enhancing generalization.
- **Hierarchical features:** Lower layers extract simple features like edges while deeper layers learn high-level patterns such as digit shapes.

2. Encoder Architecture

- **Convolutional Layers:** These extract low-level and high-level spatial features by sliding filters across the input digit images.
- **Activation Functions (ReLU):** Introduced after convolutions to introduce non-linearity, enabling the network to capture complex patterns.
- **Pooling Layers (optional):** Max-pooling or average-pooling reduces dimensionality to create a compact latent representation while retaining important features.

- **Latent Space Representation:** A compressed, information-rich encoding of the digit image. This encoded representation allows the CAE to understand typical digit structures and variations effectively.

3. Decoder Architecture

- **Transposed Convolutions (Deconvolution):** Mirror the encoder by gradually reconstructing the spatial dimension. These layers "learn" how to upsample feature maps, creating sharper reconstructions compared to dense reconstructions.
- **Upsampling Layers:** Used in combination with convolution layers to resize feature maps back to the original image dimensions.
- **Output Layer:** Produces reconstructed images with values scaled between 0 and 1 (after a sigmoid activation) to match the pixel intensity distribution of MNIST digits.

4. Benefits of Using CAEs on MNIST

- **Sharper Reconstructions:** CAEs preserve edges and strokes, allowing clearer distinction between digits with similar shapes (like "3" vs "8").
- **Robust to Noise:** Due to local pattern extraction, CAEs can act as implicit denoising systems, removing irrelevant pixel noise while retaining digit structure.
- **Dimensionality Reduction:** The encoder creates compact representations of high-dimensional image data, useful for downstream tasks like anomaly detection or clustering.
- **Generalization:** Unlike dense autoencoders, CAEs learn filters that are shift-invariant, enabling them to reconstruct digits in various positions and styles more effectively.

5. Application to the MNIST Case Study

- **Input Data:** The CAE takes 28×28 grayscale digit images as input without flattening.
- **Training Objective:** Minimize pixel-level reconstruction error (e.g., using Mean Squared Error or Binary Cross-Entropy loss).
- **Reconstruction Task:** Rebuild the image from its latent representation while retaining sharp digit boundaries.
- **Anomaly Detection:** By learning typical digit structures in latent space, CAEs can flag unusual samples with high reconstruction error, such as distorted, incomplete, or fake digits.

ALGORITHM:

1. Import Libraries: Import necessary libraries, including NumPy, Matplotlib, and relevant Keras modules. Import the MNIST dataset from Keras.
2. Load and Preprocess MNIST Data: Load the MNIST dataset, consisting of handwritten digit images. Normalize pixel values to be in the range [0, 1]. Reshape the data to add a channel dimension (for convolutional layers).
3. Define Autoencoder Model Architecture: Create an input layer for 28x28 images. Implement the encoder part with Conv2D and MaxPooling2D layers, reducing the spatial dimensions. Implement the decoder part with Conv2D and UpSampling2D layers, reconstructing the original dimensions. Compile the autoencoder using the binary cross-entropy loss and the Adam optimizer.

4. Train the Autoencoder: Train the autoencoder using the MNIST training data for 10 epochs. Use binary cross-entropy as the loss function and shuffle the training data. Validate the model's performance on the MNIST test data.
5. Generate Reconstructed Images: Use the trained autoencoder to generate reconstructed images from the MNIST test data.
6. Visualize Original and Reconstructed Images: Display original images alongside their reconstructed counterparts for visual comparison. Show 10 pairs of images, with the top row displaying original images and the bottom row displaying reconstructed images.

CODE & OUTPUT:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.datasets import mnist
```

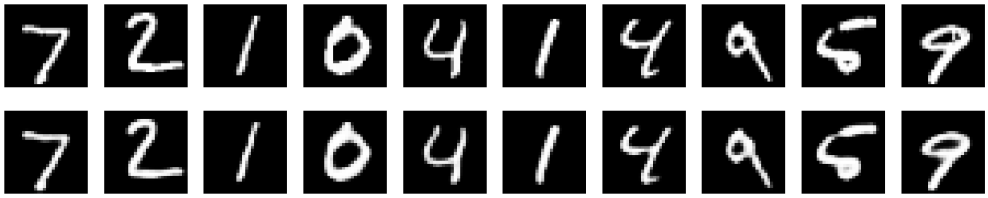
```
# Load the MNIST dataset
(x_train, _), (x_test, _) = mnist.load_data()

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

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)
```

```
def build_autoencoder(input_shape, latent_dim):
    input_img = layers.Input(shape=input_shape)  # (28, 28, 1)

    # Encoder
    x = layers.Conv2D(32, (3,3), activation='relu', padding='same')(input_img)
    x = layers.MaxPooling2D((2,2), padding='same')(x)
    x = layers.Conv2D(16, (3,3), activation='relu', padding='same')(x)
    encoded = layers.MaxPooling2D((2,2), padding='same')(x)

    # Decoder
    x = layers.Conv2D(16, (3,3), activation='relu', padding='same')(encoded)
    x = layers.UpSampling2D((2,2))(x)

autoencoder = build_autoencoder((28, 28, 1), latent_dim=64)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

```
autoencoder.fit(x_train, x_train,
                epochs=10,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test))
```

```
Epoch 1/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 111s 229ms/step - loss: 0.2168 - val_loss: 0.0802
Epoch 2/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 143s 232ms/step - loss: 0.0794 - val_loss: 0.0757
Epoch 3/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 139s 226ms/step - loss: 0.0756 - val_loss: 0.0734
Epoch 4/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 142s 225ms/step - loss: 0.0737 - val_loss: 0.0719
Epoch 5/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 105s 224ms/step - loss: 0.0726 - val_loss: 0.0710
Epoch 6/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 106s 226ms/step - loss: 0.0715 - val_loss: 0.0702
Epoch 7/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 142s 227ms/step - loss: 0.0709 - val_loss: 0.0697
Epoch 8/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 142s 226ms/step - loss: 0.0702 - val_loss: 0.0692
Epoch 9/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 105s 223ms/step - loss: 0.0698 - val_loss: 0.0689
Epoch 10/10
469/469 ━━━━━━━━━━━━━━━━━━━━ 146s 232ms/step - loss: 0.0693 - val_loss: 0.0686
<keras.src.callbacks.history.History at 0x7fdded61dee0>
```

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

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

plt.show()
```



CASE STUDY:

Case Study 1: Handwritten Digit Reconstruction (CAE on MNIST)

- **Problem**: Traditional autoencoders using dense layers struggle to reconstruct handwritten digits clearly, often leading to blurred outputs and loss of important stroke details.
- **CAE Application:**
  - **Input Layer**: 28×28 pixel handwritten digits from the MNIST dataset.
  - **Encoder**: Convolutional layers to capture edges, curves, and local structures of digits.
  - **Decoder**: Transposed convolutions and upsampling layers to reconstruct digits with minimal loss.
- **Outcome**: Improved digit reconstruction quality with sharper boundaries, effective noise reduction, and enhanced ability to differentiate visually similar digits like "3" and "8."

Case Study 2: Anomaly Detection in Digits (CAE on MNIST)

- **Problem**: Detecting unusual or malformed handwritten digits (e.g., distorted "7" or incomplete "4") is difficult with traditional approaches due to high intra-class variability.
- **CAE Application:**
  - **Input Layer:** Normal MNIST digits passed through the CAE model.
  - **Latent Space:** CAE learns compressed representations of typical digit patterns.
  - **Reconstruction**: Digits deviating from normal patterns show higher reconstruction error.
- **Outcome**: CAEs effectively identify anomalies based on reconstruction error, enabling robust detection of irregular handwriting, useful for fraud-resistant digit recognition and educational assessment tools.

LEARNING OUTCOME:

EXPERIMENT-7

AIM : Implementation of RNN model for Stock Price Prediction in Python

THEORY:

1. Introduction: Stock price prediction is an important task in financial machine learning. Since stock data is sequential, traditional machine learning models often fail to capture temporal dependencies. Recurrent Neural Networks (RNNs) are designed for sequential/time-series data, making them well- suited for stock price forecasting.

2. Why RNN for Stock Prediction?

- Stock prices form a time series where today's price is related to past values.
- RNNs contain a feedback loop that stores past information in a hidden state.
- This makes them capable of learning temporal patterns and dependencies in financial data.

3. Working of RNN: At each time step:

- Input: stock price (or multiple features such as Open, High, Low, Volume).
- Hidden state: carries information from previous steps (acts as memory).
- Output: predicted price (next day or next step).

4. Types of RNN Architectures

(a) One-to-One

- Single input → Single output.
- Example: Simple feedforward network (not typical for time series).

(b) One-to-Many

- Single input → Sequence of outputs.
- Example: Generating a stock trend sequence from a single market signal.

(c) Many-to-One

- Sequence of inputs → Single output.
- Most common in stock price prediction:
    - Past 60 days (inputs) → Next day's stock price (output).

(d) Many-to-Many

- Sequence of inputs → Sequence of outputs.
- Example: Predicting the entire future stock trend day by day.

5. Types of RNN Models

(a) Vanilla RNN (SimpleRNN)

- Basic RNN layer with hidden states.
- Can model short dependencies but struggles with long-term memory.

(b) LSTM (Long Short-Term Memory)

- Advanced RNN that solves vanishing gradient problem.
- Uses input, forget, and output gates to control memory.
- Most popular for stock prediction.

(c) GRU (Gated Recurrent Unit)

- Simplified version of LSTM.
- Uses update and reset gates.
- Faster and requires fewer parameters.

(d) Bidirectional RNN

- Processes sequences in forward and backward directions.
- More accurate for text/speech, less common in stock prediction.

6. Layers Used in Stock Price RNN

1. *Input Layer*
   - Accepts data in format (time_steps, features).
   - Example: past 60 days (time_steps = 60, features = 1 → closing price).
2. *Recurrent Layer*
   - SimpleRNN / LSTM / GRU layers capture temporal dependencies.
   - Units (neurons) define learning capacity.
3. *Dropout Layer*
   - Prevents overfitting by randomly dropping connections.
   - Example: Dropout(0.2) means 20% neurons ignored per iteration.
4. *Dense (Fully Connected) Layer*
   - Final prediction layer.
   - Example: Dense(1) → predicts a single stock price.

7. Implementation Workflow

1. Collect Data → Download stock prices (Yahoo Finance API).
2. Preprocess Data → Normalize & create sequences using sliding windows.
3. Build Model → Stack recurrent + dropout + dense layers.
4. Train Model → Minimize error with Adam optimizer & MSE loss.
5. Evaluate → Predict on test data, inverse-transform to original scale.
6. Visualize → Plot actual vs predicted prices.

ALGORITHM:

Import required Python libraries.

1. Download stock data from Yahoo Finance.
2. Select closing prices and normalize data using MinMaxScaler.
3. Create training sequences with a 60-day look-back window.
4. Reshape data into 3D format for LSTM input.
5. Build an RNN model with LSTM, Dropout, and Dense layers.
6. Compile the model with Adam optimizer and MSE loss.
7. Train the model on historical stock data.
8. Prepare test data and create sequences for prediction.
9. Predict stock prices using the trained model.
10. Inverse scale predictions back to original prices.
11. Plot historical, actual, and predicted stock prices with different colors.

CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense
from sklearn.metrics import mean_squared_error
import yfinance as yf
```

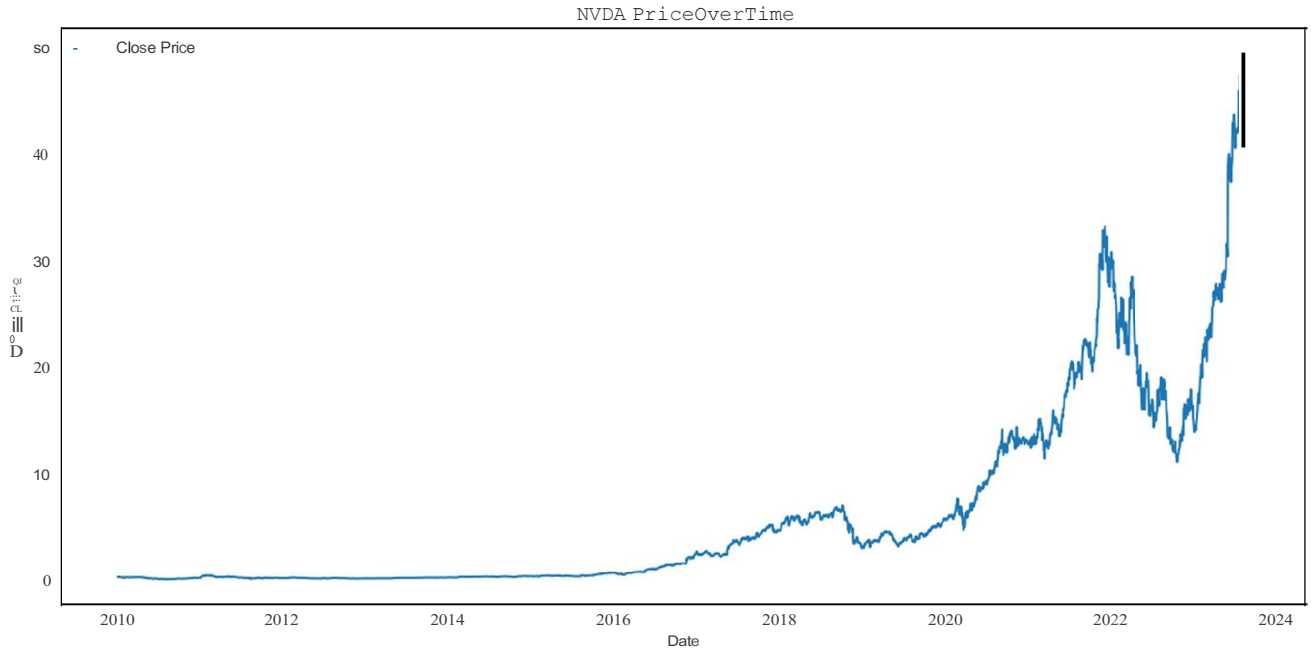
Double-click (or enter) to edit

```
# Download NVDA data from Yahoo Finance
df = yf.download('NVDA', start='2010-01-01', end='2023-09-01', progress=False)

# Keep only 'Date' and 'Close' columns
df = df[['Close']]
df.index.name = 'Date'
```

```
/tmp/ipython-input-3549399384.py:2: FutureWarning: YF.download() has changed argumer
  df = yf.download('NVDA', start='2010-01-01', end='2023-09-01', progress=False)
```

```
plt.figure(figsize=(15 ,7))
plt.plot(df.index, df['Close'], label='Close Price')
plt.title('NVDA Price Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```


```
plt.figure(figsize=(15 ,7))
plt.plot(df.index, df['Close'], label='Close Price')
plt.title('NVDA Price Over Time')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



[16]
```
scaler= MinMaxScaler(feature_range=(0, 1))
df_scaled = scaler.fit_transform(df[['Close']])
```

[17]
```
def create_dataset(data, look_back=1):
    X, y = [], []
    for i in range(len(data) - look_back):
        X.append(data[i:(i + look_back), 0])
        y.append(data[i + look_back, 0])
    return np.array(X), np.array(y)

look_back = 30
X, y = create_dataset(df_scaled, look_back)
X = np.reshape(X, (X.shape[0], 1, X.shape[1]))
```

[18]
```
split_ratio = 0.8
split_index = int(split_ratio * len(X))
X_train, X_test, y_train, y_test = X[:split_index], X[split_index:], y[:split_index], y[split_index:]
```

[19]
```
model= Sequential()
model.add(SimpleRNN(units=50, input_shape=(1, look_back), activation='relu'))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error')
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an 'input
  super().__init__(**kwargs)
```

[20]
```
model.fit(X_train, y_train, epochs=50, batch_size=32)
```

```
86/86 ---------- 0s 3ms/step - loss: 1.2866e-05
Epoch 23/50
```

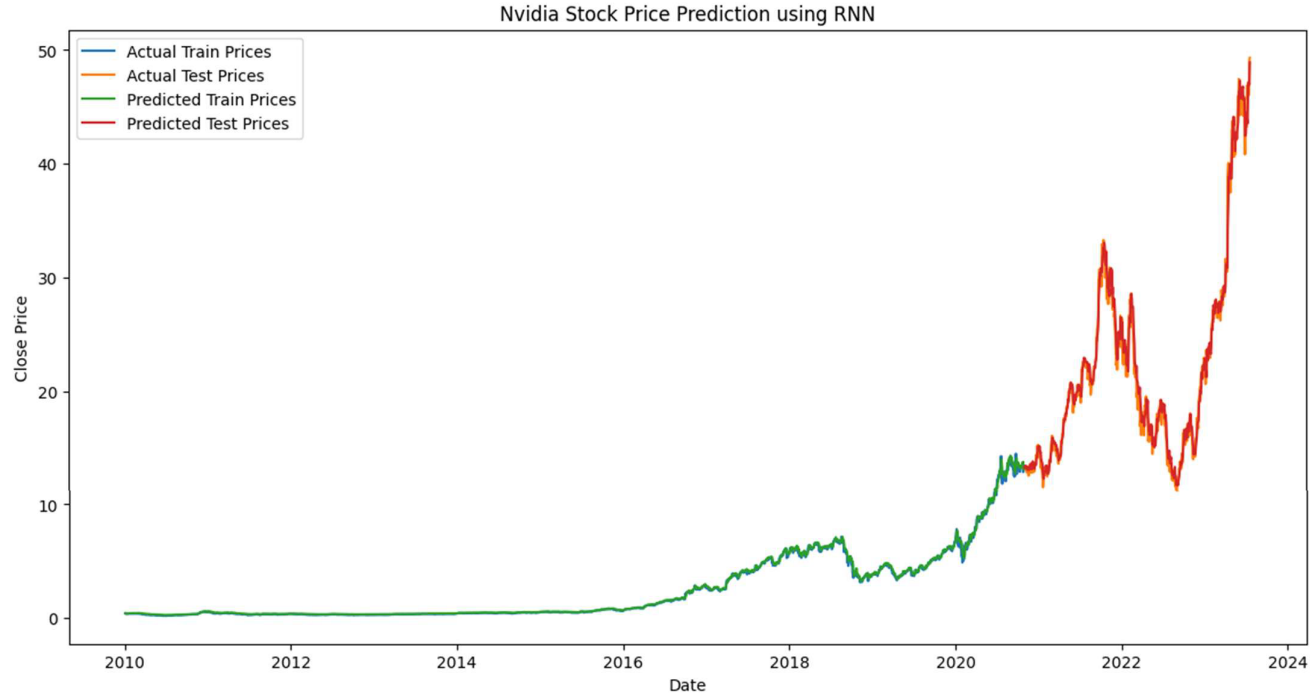
[21] ✓ 0s

```
train_predictions = model.predict(X_train)
test_predictions = model.predict(X_test)

train_predictions = scaler.inverse_transform(np.reshape(train_predictions, (-1, 1)))
test_predictions = scaler.inverse_transform(np.reshape(test_predictions, (-1, 1)))

y_train = scaler.inverse_transform(np.reshape(y_train, (-1, 1)))
y_test = scaler.inverse_transform(np.reshape(y_test, (-1, 1)))
```

```
86/86 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step
22/22 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step
```

[22]

```
plt.figure(figsize=(14, 7))
plt.plot(df.index[:len(y_train)], y_train, label='Actual Train Prices')
plt.plot(df.index[len(y_train):len(y_train) + len(y_test)], y_test, label='Actual Test Prices')
plt.plot(df.index[:len(y_train)], train_predictions, label='Predicted Train Prices')
plt.plot(df.index[len(y_train):len(y_train) + len(y_test)], test_predictions, label='Predicted Test Prices')
plt.title('Nvidia Stock Price Prediction using RNN')
plt.xlabel('Date')
plt.ylabel('Close Price')
plt.legend()
plt.show()
```



[25] ✓ 0s

```
train_rmse = np.sqrt(mean_squared_error(y_train, train_predictions))
test_rmse = np.sqrt(mean_squared_error(y_test, test_predictions))

print(f'Training RMSE: {train_rmse:.2f}')
print(f'Testing RMSE: {test_rmse:.2f}')
```

```
Training RMSE: 0.15
Testing RMSE: 1.00
```

LEARNING OUTCOMES:

EXPERIMENT-8

AIM : Implementation of LSTM model for Weather Prediction in Python

THEORY

1. Introduction

Long Short-Term Memory (LSTM) is an enhanced version of the Recurrent Neural Network (RNN) designed by Hochreiter and Schmidhuber. LSTMs can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting. Unlike traditional RNNs which use a single hidden state passed through time, LSTMs introduce a memory cell that holds information over extended periods addressing the challenge of learning long-term dependencies.

2. Problem with Long-Term Dependencies in RNN

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. However they often face challenges in learning long-term dependencies where information from distant time steps becomes crucial for making accurate predictions for current state. This problem is known as the vanishing gradient or exploding gradient problem.

Vanishing Gradient: When training a model over time, the gradients which help the model learn can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.

Exploding Gradient: Sometimes gradients can grow too large causing instability. This makes it difficult for the model to learn properly as the updates to the model become erratic and unpredictable.

Both of these issues make it challenging for standard RNNs to effectively capture long-term dependencies in sequential data.

3. LSTM Architecture

LSTM architecture involves the memory cell which is controlled by three gates:

Input gate: Controls what information is added to the memory cell.

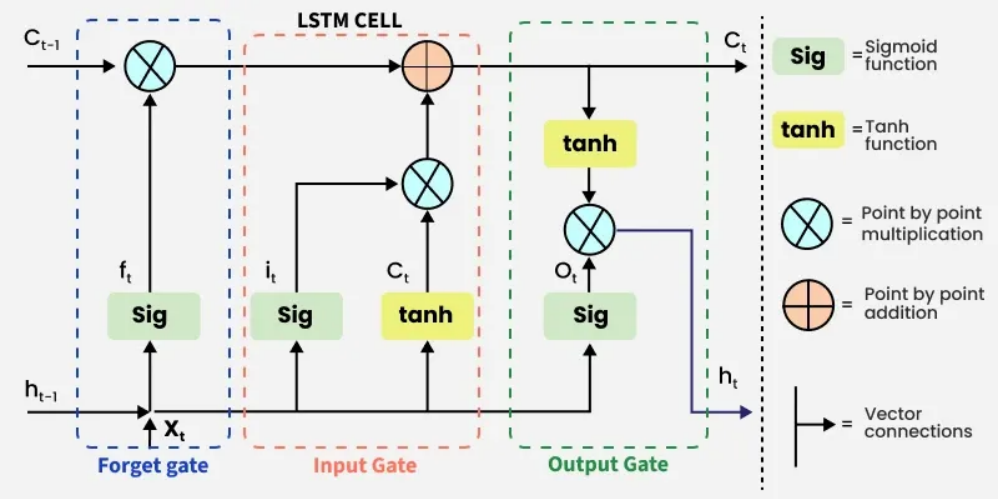
Forget gate: Determines what information is removed from the memory cell.

Output gate: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

4. Working of LSTM

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells. Information is retained by the cells and the memory manipulations are done by the gates. There are three gates:



(a) Forget Gate

The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x_t (input at the particular time) and h_{t-1} (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through sigmoid activation function which gives output in range of. If for a particular cell state the output is 0 or near to 0, the piece of information is forgotten and for output of 1 or near to 1, the information is retained for future use.

The equation for the forget gate is:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Where:

- W_f represents the weight matrix associated with the forget gate.
- $[h_{t-1}, x_t]$ denotes the concatenation of the current input and the previous hidden state.

- b_f is the bias with the forget gate.
- σ is the sigmoid activation function.

(b) Input Gate

The addition of useful information to the cell state is done by the input gate. First the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs h_{t-1} and x_t. Then, a vector is created using tanh function that gives an output from -1 to +1 which contains all the possible values from h_{t-1} and x_t. At last the values of the vector and the regulated values are multiplied to obtain the useful information.

The equation for the input gate is:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\hat{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

We multiply the previous state by f_t effectively filtering out the information we had decided to ignore earlier. Then we add $i_t \odot \hat{C}_t$ which represents the new candidate values scaled by how much we decided to update each state value:

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t$$

where $\odot$ denotes element-wise multiplication.

(c) Output Gate

The output gate is responsible for deciding what part of the current cell state should be sent as the hidden state (output) for this time step. First, the gate uses a sigmoid function to determine which information from the current cell state will be output. This is done using the previous hidden state h_{t-1} and the current input x_t:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Next, the current cell state C_t is passed through a tanh activation to scale its values between -1 and +1. Finally, this transformed cell state is multiplied element-wise with o_t to produce the hidden state h_t:

$$h_t = o_t \odot \tanh(C_t)$$

Here:

- o_t is the output gate activation.
- C_t is the current cell state.
- $\odot$ represents element-wise multiplication.

This hidden state h_t is then passed to the next time step and can also be used for generating the output of the network.

5. Types of RNN Architectures

(a) One-to-One

Single input → Single output.

Example: Simple feedforward network (not typical for sequential tasks).

(b) One-to-Many

Single input → Sequence of outputs.

Example: Image captioning where a single image generates a sequence of words.

(c) Many-to-One

Sequence of inputs → Single output.

Most common in classification tasks.

Example: Sentiment analysis where a sequence of words produces a single sentiment label.

(d) Many-to-Many

Sequence of inputs → Sequence of outputs.

Example: Language translation where input sequence is converted to output sequence.

6. Applications

Time-Series Forecasting: Stock price prediction, weather forecasting.

Natural Language Processing: Language translation, text generation, sentiment analysis.

Speech Recognition: Converting speech to text by capturing temporal patterns.

Video Processing: Analyzing sequential frames in videos.

SOURCE CODE:

```
# Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Input
```

More actions

```
# Load the dataset
df = pd.read_csv('export.csv')
print(df.head())
```

```
                 date  tavg  tmin  tmax  prcp  snow  wdir  wspd  wpgt  \
0  2024-08-07 00:00:00  28.0  26.4  35.2   0.3   NaN   NaN   5.4   NaN
1  2024-08-08 00:00:00  29.4  25.4  32.8  21.1   NaN   NaN   5.6   NaN
2  2024-08-09 00:00:00  29.4  26.6  34.6   3.0   NaN   NaN   4.7   NaN
3  2024-08-10 00:00:00  29.3  25.8  34.8  17.0   NaN   NaN   4.5   NaN
4  2024-08-11 00:00:00  27.8  26.0  32.8   3.0   NaN   NaN   2.7   NaN


     pres  tsun
0   999.4   NaN
1  1000.9   NaN
2  1001.9   NaN
3  1001.2   NaN
4  1000.1   NaN
```

```
# Select the features for the LSTM model
features = ['tavg', 'tmin', 'tmax']
data = df[features].values

# Scale the data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)

# Create sequences for LSTM
def create_sequences(data, look_back):
    X, y = [], []
    for i in range(len(data) - look_back):
        X.append(data[i:(i + look_back), :])

        y.append(data[i + look_back, 0])
    return np.array(X), np.array(y)

look_back = 22

# Create sequences
X, y = create_sequences(scaled_data, look_back)

train_size = int(len(X) * 0.7)
test_size = len(X) - train_size
X_train, X_test = X[0:train_size,:], X[train_size:len(X),:]
y_train, y_test = y[0:train_size], y[train_size:len(y)]


print("Shape of X_train:", X_train.shape)
print("Shape of y_train:", y_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of y_test:", y_test.shape)
```

```
Shape of X_train: (251, 22, 3)
Shape of y_train: (251,)
Shape of X_test: (108, 22, 3)
Shape of y_test: (108,)
```

```
# Define the LSTM model
model = Sequential()
model.add(Input(shape=(look_back, len(features)))) # Update input shape to match the number of features
model.add(LSTM(50)) # Add LSTM layer with 50 units
model.add(Dense(1)) # Add Dense output layer with 1 unit

# Compile the model
model.compile(optimizer='adam', loss='mean_squared_error')
```

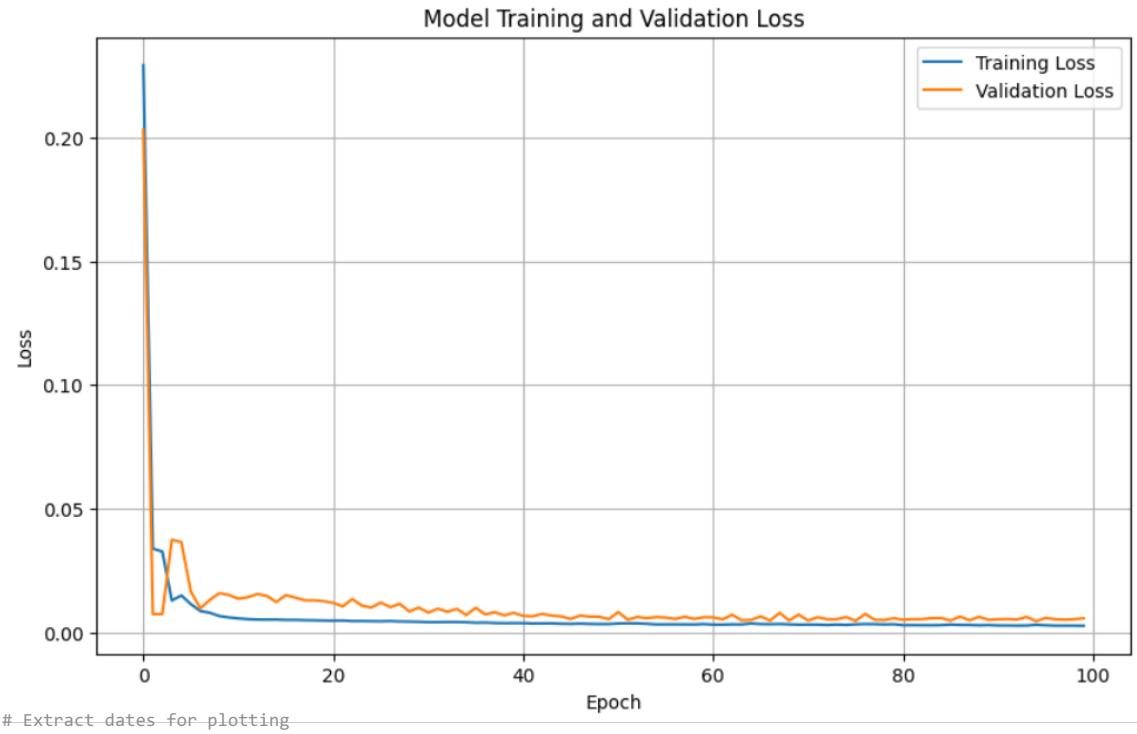
```
Model: "sequential_14"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━┓
┃ Layer (type)                         ┃ Output Shape                ┃         Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━┩
│ lstm_14 (LSTM)                       │ (None, 50)                  │          10,800 │
├──────────────────────────────────────┼─────────────────────────────┼─────────────────┤
│ dense_14 (Dense)                     │ (None, 1)                   │              51 │
└──────────────────────────────────────┴─────────────────────────────┴─────────────────┘
 Total params: 10,851 (42.39 KB)
 Trainable params: 10,851 (42.39 KB)
 Non-trainable params: 0 (0.00 B)
```

```
# Train the model with validation split
history = model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=2, validation_split=0.2) # Add validation_split

Epoch 1/100
7/7 - 3s - 446ms/step - loss: 0.2292 - val_loss: 0.2031
Epoch 2/100
7/7 - 1s - 99ms/step - loss: 0.0339 - val_loss: 0.0073
Epoch 3/100
7/7 - 0s - 25ms/step - loss: 0.0326 - val_loss: 0.0074
Epoch 4/100
7/7 - 0s - 25ms/step - loss: 0.0128 - val_loss: 0.0374
Epoch 5/100
7/7 - 0s - 25ms/step - loss: 0.0149 - val_loss: 0.0365
Epoch 6/100
7/7 - 0s - 24ms/step - loss: 0.0114 - val_loss: 0.0165
Epoch 7/100
7/7 - 0s - 24ms/step - loss: 0.0086 - val_loss: 0.0097
Epoch 8/100
7/7 - 0s - 27ms/step - loss: 0.0079 - val_loss: 0.0131
Epoch 9/100
7/7 - 0s - 24ms/step - loss: 0.0066 - val_loss: 0.0158
Epoch 10/100
7/7 - 0s - 25ms/step - loss: 0.0060 - val_loss: 0.0151
Epoch 11/100
7/7 - 0s - 24ms/step - loss: 0.0056 - val_loss: 0.0136
Epoch 12/100
7/7 - 0s - 24ms/step - loss: 0.0053 - val_loss: 0.0142
Epoch 13/100
7/7 - 0s - 25ms/step - loss: 0.0052 - val_loss: 0.0155
Epoch 14/100
7/7 - 0s - 27ms/step - loss: 0.0051 - val_loss: 0.0147
Epoch 15/100
7/7 - 0s - 24ms/step - loss: 0.0051 - val_loss: 0.0122
Epoch 16/100
7/7 - 0s - 25ms/step - loss: 0.0050 - val_loss: 0.0150
Epoch 17/100
7/7 - 0s - 26ms/step - loss: 0.0050 - val_loss: 0.0140
Epoch 18/100
7/7 - 0s - 26ms/step - loss: 0.0049 - val_loss: 0.0129
Epoch 19/100
7/7 - 0s - 27ms/step - loss: 0.0048 - val_loss: 0.0129
Epoch 20/100
7/7 - 0s - 26ms/step - loss: 0.0048 - val_loss: 0.0126
Epoch 21/100
7/7 - 0s - 26ms/step - loss: 0.0047 - val_loss: 0.0119
Epoch 22/100
7/7 - 0s - 24ms/step - loss: 0.0048 - val_loss: 0.0105
Epoch 23/100
7/7 - 0s - 25ms/step - loss: 0.0045 - val_loss: 0.0134
Epoch 24/100
7/7 - 0s - 26ms/step - loss: 0.0046 - val_loss: 0.0108
Epoch 25/100
7/7 - 0s - 26ms/step - loss: 0.0045 - val_loss: 0.0100
Epoch 26/100
7/7 - 0s - 24ms/step - loss: 0.0045 - val_loss: 0.0120
Epoch 27/100
7/7 - 0s - 25ms/step - loss: 0.0046 - val_loss: 0.0102
Epoch 28/100
7/7 - 0s - 25ms/step - loss: 0.0044 - val_loss: 0.0115
Epoch 29/100
7/7 - 0s - 24ms/step - loss: 0.0043 - val_loss: 0.0084
```

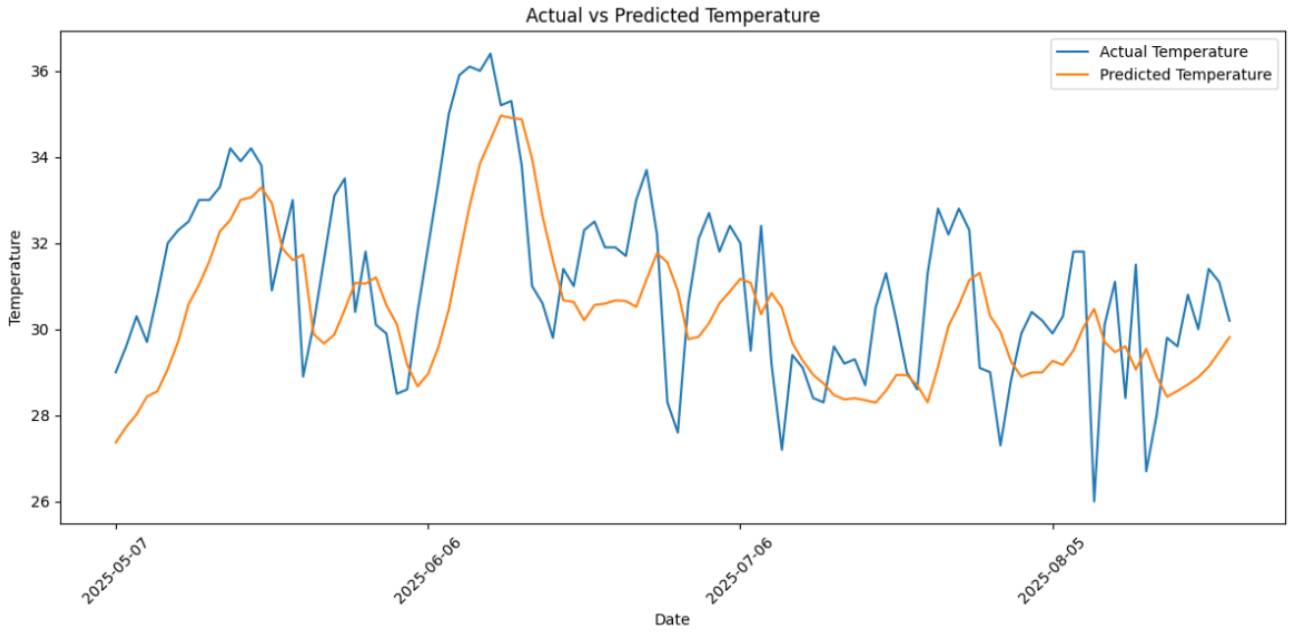
```
y_pred = model.predict(X_test)
dummy_array_pred = np.zeros((len(y_pred), len(features)))
dummy_array_pred[:, 0] = y_pred[:, 0]
y_pred_inv = scaler.inverse_transform(dummy_array_pred)[:, 0] # Inverse transform and select the first column

dummy_array_test = np.zeros((len(y_test), len(features)))
dummy_array_test[:, 0] = y_test # y_test is already 1D
y_test_inv = scaler.inverse_transform(dummy_array_test)[:, 0] # Inverse transform and select the first column

# Calculate and print RMSE
rmse = np.sqrt(mean_squared_error(y_test_inv, y_pred_inv))
print(f"Root Mean Squared Error (RMSE) on Test Set: {rmse}")
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 69ms/step
Root Mean Squared Error (RMSE) on Test Set: 1.9542549838547711
```

```
# Plot training and validation loss
plt.figure(figsize=(10, 6))
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```




```
# Extract dates for plotting
test_dates = df['date'].iloc[train_size + look_back:].reset_index(drop=True)
test_dates = pd.to_datetime(test_dates)

# Plot actual vs predicted temperatures
plt.figure(figsize=(12, 6))
plt.plot(test_dates, y_test_inv, label='Actual Temperature')
plt.plot(test_dates, y_pred_inv, label='Predicted Temperature')
plt.title('Actual vs Predicted Temperature')
plt.xlabel('Date')
plt.ylabel('Temperature')
plt.legend()

# Set x-axis ticks to show monthly intervals
plt.xticks(test_dates[::30], rotation=45)
plt.tight_layout()
plt.show()
```



LEARNING OUTCOME:

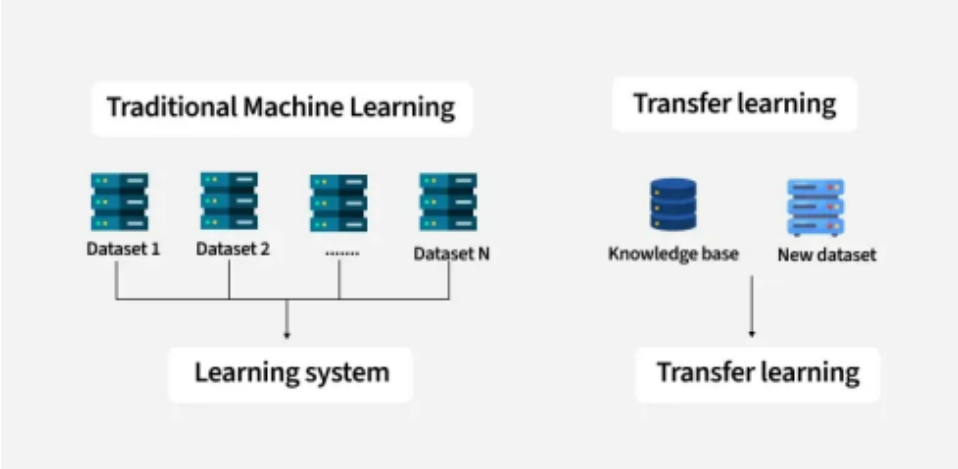
EXPERIMENT - 9

AIM : Implementation of transfer learning using pre-trained model (MobileNet V2) for image classification in python.

THEORY:

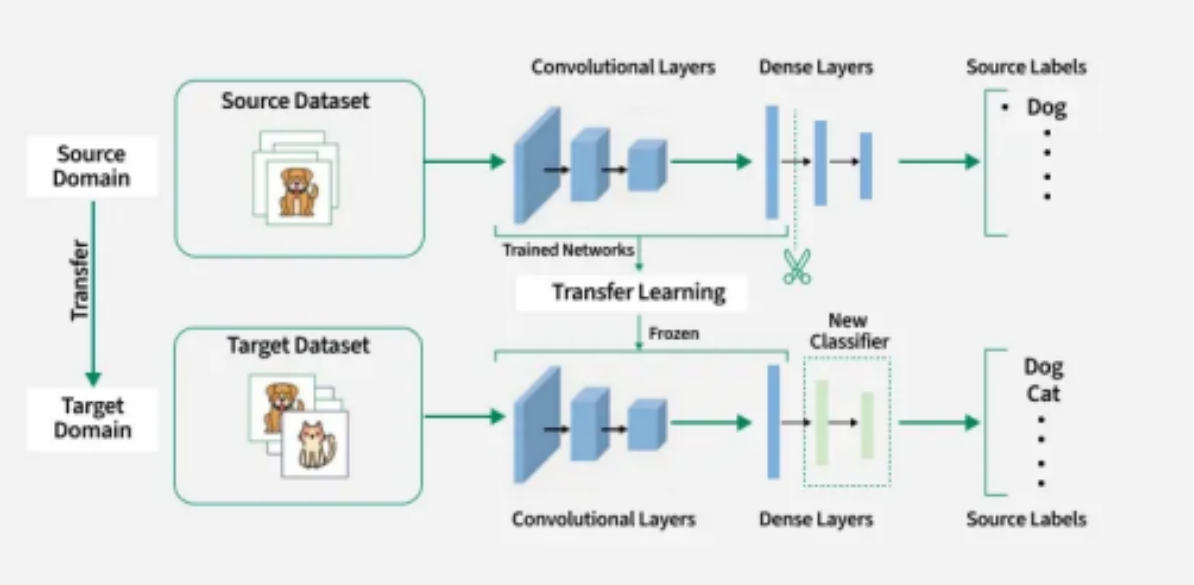
1. INTRODUCTION

Transfer Learning is an advanced machine learning approach where a model developed for one specific task is reused as the starting point for a second, related task. Instead of training a model from scratch, Transfer Learning leverages the knowledge gained from solving one problem and applies it to another with similar characteristics. This is especially advantageous when the new task has limited labeled data or requires faster convergence. Commonly used in computer vision, natural language processing, and speech recognition, Transfer Learning enables the reuse of deep neural network architectures like VGG, ResNet, BERT, and GPT.



2. IMPORTANCE OF TRANFER LEARNING

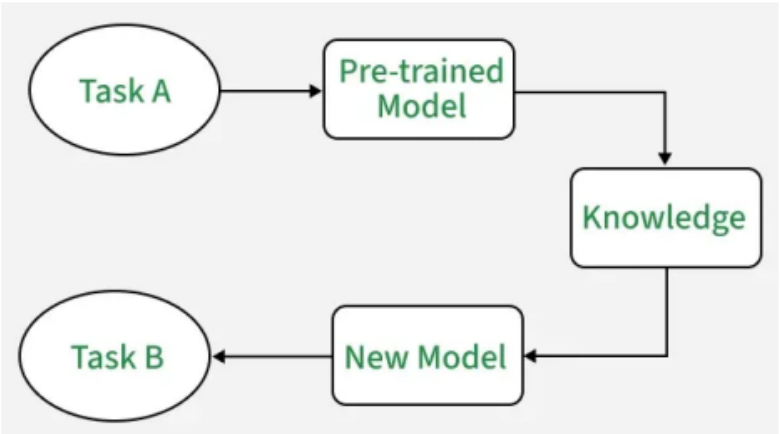
- **Limited Data Handling:** In many real-world problems, collecting large labeled datasets is challenging. Transfer Learning allows the use of pre-trained models that have already learned generalized patterns, significantly reducing the need for extensive data.
- **Improved Accuracy:** Since pre-trained models are trained on large and diverse datasets, they provide better feature extraction capabilities, leading to faster training and higher performance in new tasks.
- **Reduced Computational Cost:** Training deep learning models from scratch is resource-intensive. Transfer Learning minimizes both time and cost by reusing existing models and weights.
- **High Adaptability:** It enables models to be fine-tuned for a wide range of related applications—such as adapting an image classification model trained on ImageNet for medical imaging or satellite analysis—demonstrating strong generalization ability.



3. WORKING OF TRANFER LEARNING

The process of Transfer Learning typically involves the following stages:

- **Selecting a Pre-Trained Model:** Begin with a model already trained on a large benchmark dataset (e.g., ImageNet or COCO). This model has learned fundamental patterns and low- to high-level features.
- **Using as Base Model:** The pre-trained network is used as the base model. Early layers generally capture generic features like edges and shapes, while deeper layers represent task-specific abstractions.
- **Freezing and Transferring Layers:** Certain layers of the base model are frozen to retain the learned weights. The final layers are modified or replaced to suit the new task while keeping earlier knowledge intact.
- **Fine-Tuning:** The model is then fine-tuned using the new dataset, allowing it to adjust its parameters to the specific domain or task. This combination of preserved knowledge and targeted training ensures efficiency and accuracy.



CODE:

```
# Step 1: Preparing the Dataset
!pip install tensorflow
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

print("Train images shape:", train_images.shape)
print("Test images shape:", test_images.shape)
```

```
Collecting tensorflow
  Downloading tensorflow-2.20.0-cp312-cp312-manylinux_2_17_x86_64.manylinux2014_x8
Requirement already satisfied: absl-py>=1.0.0 in /usr/local/lib/python3.12/dist-pa
Collecting astunparse>=1.6.0 (from tensorflow)
  Downloading astunparse-1.6.3-py2.py3-none-any.whl.metadata (4.4 kB)
Collecting flatbuffers>=24.3.25 (from tensorflow)
  Downloading flatbuffers-25.9.23-py2.py3-none-any.whl.metadata (875 bytes)
```

```
# Convert grayscale (28x28) to RGB (28x28x3) by duplicating channels
train_images = np.stack([train_images] * 3, axis=-1) / 255.0
test_images = np.stack([test_images] * 3, axis=-1) / 255.0

# Resize images to 32x32 for MobileNetV2
train_images = tf.image.resize(train_images, [32, 32])
test_images = tf.image.resize(test_images, [32, 32])

print("New train shape:", train_images.shape)
print("New test shape:", test_images.shape)
```

```
New train shape: (60000, 32, 32, 3)
New test shape: (10000, 32, 32, 3)
```

```
# One-hot encode labels
train_labels = to_categorical(train_labels, 10)
test_labels = to_categorical(test_labels, 10)

print("Example encoded label:", train_labels[0])
```

```
Example encoded label: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.layers import Input, GlobalAveragePooling2D, Dense
from tensorflow.keras.models import Model

# Load pre-trained MobileNetV2 model (without top)
base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
base_model.trainable = False  # Freeze base layers

# Add custom classification head
inputs = Input(shape=(32, 32, 3))
x = base_model(inputs, training=False)
x = GlobalAveragePooling2D()(x)
outputs = Dense(10, activation='softmax')(x)

model = Model(inputs, outputs)
model.summary()
```

```
/tmp/ipython-input-2721919624.py:6: UserWarning: `input_shape` is undefined or non-square, or `rows`
  base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobil
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Model: "functional"
```

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 32, 32, 3)	0
mobilenetv2_1.00_224 (Functional)	(None, 1, 1, 1280)	2,257,984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense (Dense)	(None, 10)	12,810

```
Total params: 2,270,794 (8.66 MB)
Trainable params: 12,810 (50.04 KB)
Non-trainable params: 2,257,984 (8.61 MB)
```

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

# Train model
history1 = model.fit(train_images, train_labels, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 24s 15ms/step - accuracy: 0.4383 - loss: 1.8364 - val_accuracy: 0.5993 - val_loss: 1.2990
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6113 - loss: 1.2548 - val_accuracy: 0.6370 - val_loss: 1.1403
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6373 - loss: 1.1335 - val_accuracy: 0.6553 - val_loss: 1.0720
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6500 - loss: 1.0742 - val_accuracy: 0.6597 - val_loss: 1.0367
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.6557 - loss: 1.0406 - val_accuracy: 0.6643 - val_loss: 1.0150
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6618 - loss: 1.0149 - val_accuracy: 0.6677 - val_loss: 1.0014
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6621 - loss: 1.0092 - val_accuracy: 0.6693 - val_loss: 0.9898
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6648 - loss: 1.0007 - val_accuracy: 0.6726 - val_loss: 0.9824
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.6679 - loss: 0.9885 - val_accuracy: 0.6724 - val_loss: 0.9775
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.6667 - loss: 0.9921 - val_accuracy: 0.6738 - val_loss: 0.9722
```

```
# Unfreeze base model for fine-tuning
base_model.trainable = True

# Freeze first 100 layers
for layer in base_model.layers[:100]:
    layer.trainable = False

# Compile with lower learning rate
model.compile(optimizer=tf.keras.optimizers.Adam(1e-5),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Fine-tune model
history2 = model.fit(train_images, train_labels, epochs=5, validation_split=0.2)
```

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 59s 35ms/step - accuracy: 0.2296 - loss: 11.1687 - val_accuracy: 0.1636 - val_loss: 14.6459
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.4837 - loss: 2.9717 - val_accuracy: 0.3301 - val_loss: 2.5147
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.5944 - loss: 1.8080 - val_accuracy: 0.6367 - val_loss: 1.2006
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.6825 - loss: 1.2592 - val_accuracy: 0.7784 - val_loss: 0.8199
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 50s 33ms/step - accuracy: 0.7444 - loss: 0.9591 - val_accuracy: 0.8291 - val_loss: 0.6031
```

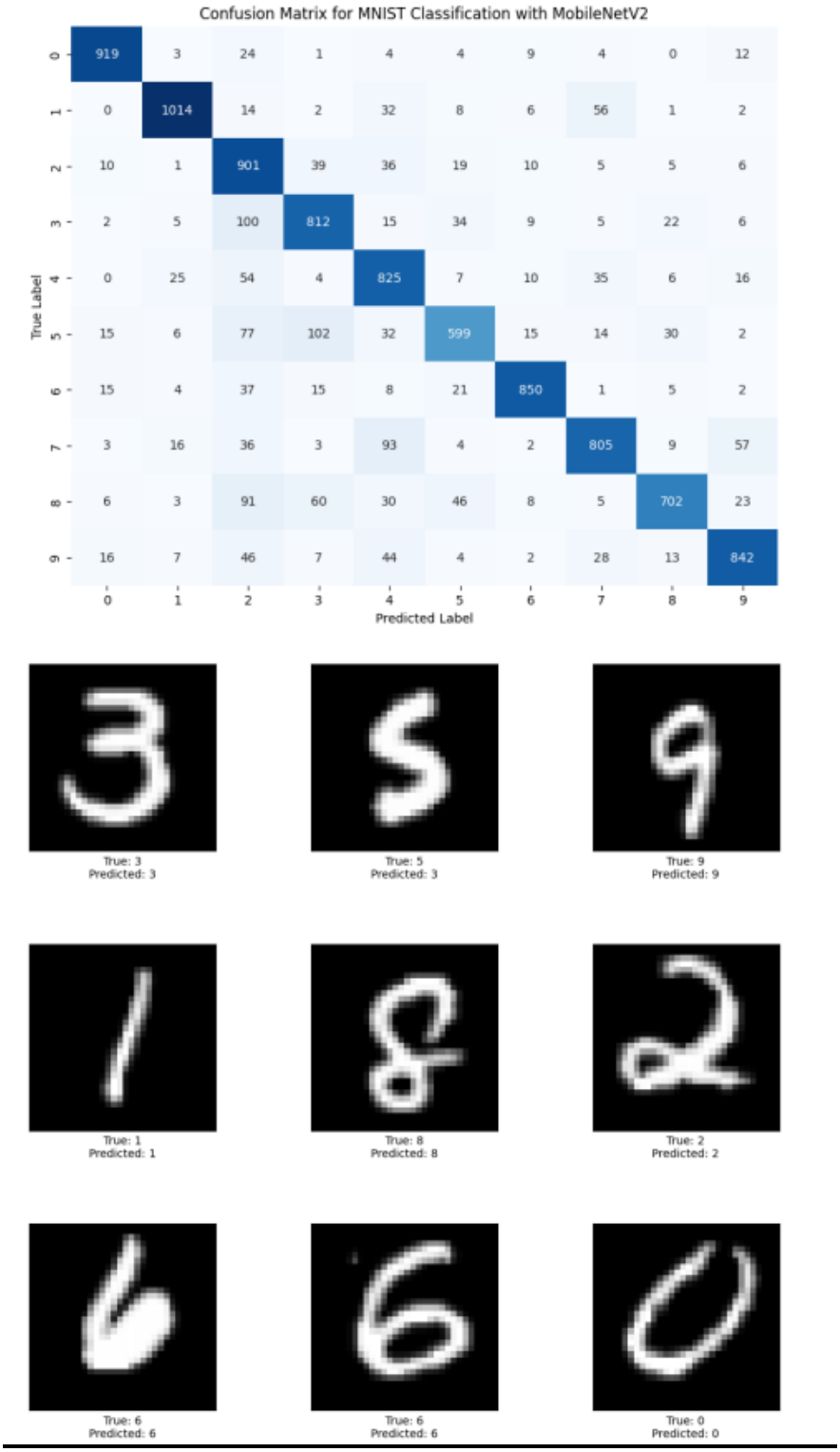
```
# Evaluate on test data
loss, accuracy = model.evaluate(test_images, test_labels)
print(f"Test loss: {loss:.4f}")
print(f"Test accuracy: {accuracy:.4f}")
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.8117 - loss: 0.6716
Test loss: 0.6333
Test accuracy: 0.8269
```

```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict on test set
test_predictions = model.predict(test_images)
test_predictions_classes = np.argmax(test_predictions, axis=1)
test_true_classes = np.argmax(test_labels, axis=1)

# Confusion Matrix
cm = confusion_matrix(test_true_classes, test_predictions_classes)

# Plot
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix for MNIST Classification with MobileNetV2")
plt.show()
```

```
# Step 7: Visualize Sample Predictions
def display_sample(sample_images, sample_labels, sample_predictions):
    fig, axes = plt.subplots(3, 3, figsize=(12, 12))
    fig.subplots_adjust(hspace=0.5, wspace=0.5)

    for i, ax in enumerate(axes.flat):
        ax.imshow(sample_images[i].reshape(32, 32), cmap='gray')
        ax.set_xlabel(f"True: {sample_labels[i]}\nPredicted: {sample_predictions[i]}")
        ax.set_xticks([])
        ax.set_yticks([])

    plt.show()

# Convert test images to grayscale for display
test_images_gray = np.dot(test_images[...,:3], [0.2989, 0.5870, 0.1140])

# Pick 9 random samples
random_indices = np.random.choice(len(test_images_gray), 9, replace=False)
sample_images = test_images_gray[random_indices]
sample_labels = test_true_classes[random_indices]
sample_predictions = test_predictions_classes[random_indices]

# Display
display_sample(sample_images, sample_labels, sample_predictions)
```

OUTPUT:



LEARNING OUTCOMES:

EXPERIMENT-10

AIM : Implementation of Transfer Learning using the pre-trained model (VGG16) on image dataset in Python

THEORY:

1. INTRODUCTION

VGG16, short for Visual Geometry Group 16, is a convolutional neural network (CNN) architecture designed for image classification. It was developed by the Visual Geometry Group at the University of Oxford.

2. ARCHITECTURE:

a) **Depth & Simplicity:**
VGG16 is characterized by its simplicity and uniform architecture. It has 16 layers, consisting mainly of 3x3 convolutional layers with small receptive fields. The uniformity of architecture (using only 3x3 convolutions) makes it easy to understand and implement.

b) **Convolutional Blocks:**
The architecture is divided into five blocks, where each block contains multiple convolutional layers followed by a max-pooling layer. The convolutional layers are configured to have a small receptive field, enabling the network to learn intricate features.

c) **Fully Connected Layers:**
The final part of the network includes three fully connected layers, followed by a softmax activation layer for classification. These fully connected layers consolidate high-level features learned by the convolutional layers for making class predictions.

Receptive Field:
The use of multiple 3x3 convolutional layers in sequence effectively simulates a larger receptive field without significantly increasing the number of parameters. For instance, two 3x3 convolutions have the same effective receptive field as a single 5x5 convolution but with fewer parameters.

Transfer Learning:
VGG16 is often used as a pre-trained model for transfer learning due to its effectiveness on various image-related tasks. The pre-trained weights learned on large datasets, such as ImageNet, can be transferred and fine-tuned for specific image classification tasks with limited data.

Applications:
VGG16 has been widely employed in computer vision tasks, including image classification, object detection, and feature extraction. Its versatility and performance have made it a popular choice in both research and practical applications.

Challenges:
While VGG16 performs well, it comes with a high computational cost due to its large number of parameters. Training and deploying VGG16 may require substantial resources, making it less suitable for resource-constrained environments

Impact:
VGG16, along with its deeper variant VGG19, played a pivotal role in the development of deeper convolutional neural network architectures. Its architecture principles influenced subsequent designs and paved the way for the development of more sophisticated models

CODE:

```
import tensorflow as tf
from google.colab import drive
import os
import matplotlib.pyplot as plt
from glob import glob
from tensorflow.keras.layers import Input, Lambda, Dense, Flatten
from tensorflow.keras.models import Model
from tensorflow.keras.applications.vgg16 import VGG16
from tensorflow.keras.preprocessing.image import ImageDataGenerator

print("TensorFlow Version:", tf.__version__)

drive.mount('/content/drive')

dataset_path = "/content/drive/MyDrive/FDL 5th Sem Practical Codes/Cotton Disease Dataset/Cotton Disease"
train_path = os.path.join(dataset_path, "train")
valid_path = os.path.join(dataset_path, "test")

IMAGE_SIZE = [224, 224]
vgg16 = VGG16(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False)

for layer in vgg16.layers:
    layer.trainable = False

folders = glob(train_path + "/*")
x = Flatten()(vgg16.output)
prediction = Dense(len(folders), activation='softmax')(x)
model = Model(inputs=vgg16.input, outputs=prediction)
model.summary()
```

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

train_datagen = ImageDataGenerator(rescale=1./255, shear_range=0.2, zoom_range=0.2, horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1./255)

training_set = train_datagen.flow_from_directory(train_path, target_size=(224, 224), batch_size=32, class_mode='categorical')
test_set = test_datagen.flow_from_directory(valid_path, target_size=(224, 224), batch_size=32, class_mode='categorical')

r = model.fit(training_set, validation_data=test_set, epochs=20, steps_per_epoch=len(training_set), validation_steps=len(test_set))

plt.figure(figsize=(8,6))
plt.plot(r.history['loss'], label='Train Loss')
plt.plot(r.history['val_loss'], label='Validation Loss')
plt.legend()
plt.title("Training vs Validation Loss")
plt.show()

plt.figure(figsize=(8,6))
plt.plot(r.history['accuracy'], label='Train Accuracy')
plt.plot(r.history['val_accuracy'], label='Validation Accuracy')
plt.legend()
plt.title("Training vs Validation Accuracy")
plt.show()
```

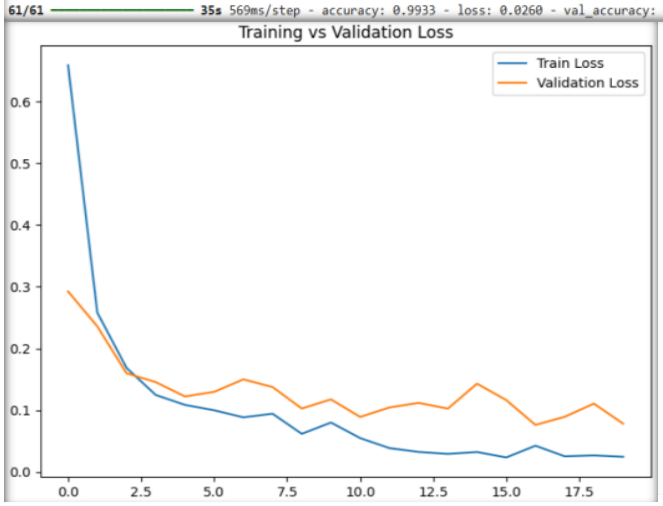
```
TensorFlow Version: 2.19.0
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount(
```

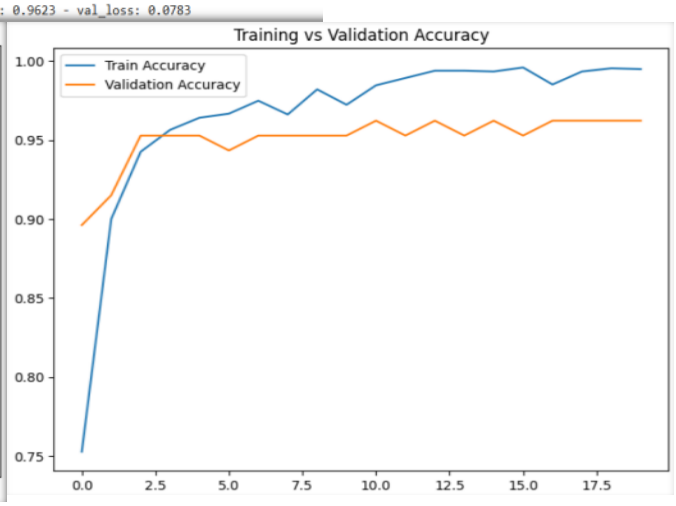
Model: "functional_1"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten_1 (Flatten)	(None, 25088)	0
dense_1 (Dense)	(None, 4)	100,356

Total params: 14,815,044 (56.51 MB)
Trainable params: 100,356 (392.02 KB)
Non-trainable params: 14,714,688 (56.13 MB)

```
Found 1951 images belonging to 4 classes.
Found 106 images belonging to 4 classes.
/usr/local/lib/python3.12/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121: UserWarning: Your `PyDataset` class
  self._warn_if_super_not_called()
Epoch 1/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 678s 11s/step - accuracy: 0.6207 - loss: 1.0813 - val_accuracy: 0.8962 - val_loss: 0.2924
Epoch 2/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 570ms/step - accuracy: 0.8903 - loss: 0.2813 - val_accuracy: 0.9151 - val_loss: 0.2359
Epoch 3/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 576ms/step - accuracy: 0.9474 - loss: 0.1688 - val_accuracy: 0.9528 - val_loss: 0.1600
Epoch 4/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 34s 561ms/step - accuracy: 0.9605 - loss: 0.1208 - val_accuracy: 0.9528 - val_loss: 0.1455
Epoch 5/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 36s 584ms/step - accuracy: 0.9670 - loss: 0.1066 - val_accuracy: 0.9528 - val_loss: 0.1223
Epoch 6/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 565ms/step - accuracy: 0.9658 - loss: 0.1079 - val_accuracy: 0.9434 - val_loss: 0.1298
Epoch 7/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 41s 570ms/step - accuracy: 0.9851 - loss: 0.0713 - val_accuracy: 0.9528 - val_loss: 0.1500
Epoch 8/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 569ms/step - accuracy: 0.9600 - loss: 0.1078 - val_accuracy: 0.9528 - val_loss: 0.1376
Epoch 9/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 578ms/step - accuracy: 0.9844 - loss: 0.0608 - val_accuracy: 0.9528 - val_loss: 0.1028
Epoch 10/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 34s 562ms/step - accuracy: 0.9726 - loss: 0.0787 - val_accuracy: 0.9528 - val_loss: 0.1176
Epoch 11/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 569ms/step - accuracy: 0.9849 - loss: 0.0565 - val_accuracy: 0.9623 - val_loss: 0.0891
Epoch 12/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 577ms/step - accuracy: 0.9927 - loss: 0.0320 - val_accuracy: 0.9528 - val_loss: 0.1045
Epoch 13/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 41s 573ms/step - accuracy: 0.9921 - loss: 0.0340 - val_accuracy: 0.9623 - val_loss: 0.1119
Epoch 14/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 34s 565ms/step - accuracy: 0.9950 - loss: 0.0268 - val_accuracy: 0.9528 - val_loss: 0.1026
Epoch 15/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 575ms/step - accuracy: 0.9944 - loss: 0.0325 - val_accuracy: 0.9623 - val_loss: 0.1431
Epoch 16/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 573ms/step - accuracy: 0.9963 - loss: 0.0267 - val_accuracy: 0.9528 - val_loss: 0.1165
Epoch 17/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 41s 574ms/step - accuracy: 0.9863 - loss: 0.0401 - val_accuracy: 0.9623 - val_loss: 0.0762
Epoch 18/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 34s 556ms/step - accuracy: 0.9936 - loss: 0.0253 - val_accuracy: 0.9623 - val_loss: 0.0894
Epoch 19/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 569ms/step - accuracy: 0.9927 - loss: 0.0319 - val_accuracy: 0.9623 - val_loss: 0.1107
Epoch 20/20
61/61 ━━━━━━━━━━━━━━━━━━━━ 35s 569ms/step - accuracy: 0.9933 - loss: 0.0260 - val_accuracy: 0.9623 - val_loss: 0.0783
```





LEARNING OUTCOMES:

EXPERIMENT – 11

AIM: Natural language processing (NLP) analysis of Restaurant reviews.

THEORY:

Natural Language Processing (NLP) for restaurant reviews involves analyzing text data to classify feedback as positive or negative using machine learning techniques. The process follows a structured workflow, starting with data import and preprocessing and ending with model evaluation.

NLP is a subfield of AI focused on enabling computers to understand and process human language. For restaurant review analysis, the aim is predictive classification—distinguishing positive from negative sentiments in reviews.

Key Steps in NLP Analysis

NLP-based restaurant review analysis classifies text into positive or negative sentiment by transforming raw reviews into numerical features and training a supervised model for prediction. The pipeline typically includes text cleaning, feature extraction with Bag of Words, model training (e.g., Random Forest), and performance evaluation using a confusion matrix.

Data and preprocessing

- Reviews are loaded from a tab-separated file where the first column contains text and the second contains labels (0 = negative, 1 = positive).
- Cleaning includes removing non-alphabetic characters, lowercasing, removing stopwords, and stemming to reduce words to their root forms, producing a normalized corpus for modeling.
- Tokenization splits text into tokens to build a consistent representation across documents as input to feature extraction methods.

Feature extraction

- Bag of Words via CountVectorizer converts the corpus into a sparse document-term matrix where each column is a unique word and each cell stores the count of that word in a review.
- Limiting the vocabulary size (e.g., max_features = 1500) focuses on the most informative words, reducing dimensionality and improving efficiency without heavy loss of signal.

Modeling

- The dataset is split into training and test sets (e.g., 75/25) to evaluate generalization on unseen data.
- A Random Forest classifier is trained on the bag-of-words features; its ensemble of trees improves robustness and allows non-linear decision boundaries for sentiment classification.

Evaluation

- Predictions on the test set are summarized with a confusion matrix containing true positives, true negatives, false positives, and false negatives to understand both accuracy and types of errors.
- Beyond accuracy, precision, recall, and related metrics derived from the confusion matrix provide a fuller view of model performance, especially with class imbalance.

SOURCE CODE:

```
[ ]
# Import libraries
import numpy as np
import pandas as pd
import re
import nltk
import matplotlib.pyplot as plt
import seaborn as sns
```

```
[ ]
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
```

```
[ ]
# Stopwords
nltk.download('stopwords')

# Load dataset
dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter='\t')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
[ ]
# Data cleaning and preprocessing
corpus = []
ps = PorterStemmer()

for i in range(0, len(dataset)):
    review = re.sub('[^a-zA-Z]', ' ', dataset['Review'][i])
    review = review.lower().split()
    review = [ps.stem(word) for word in review if word not in stopwords.words('english')]
    corpus.append(' '.join(review))
```

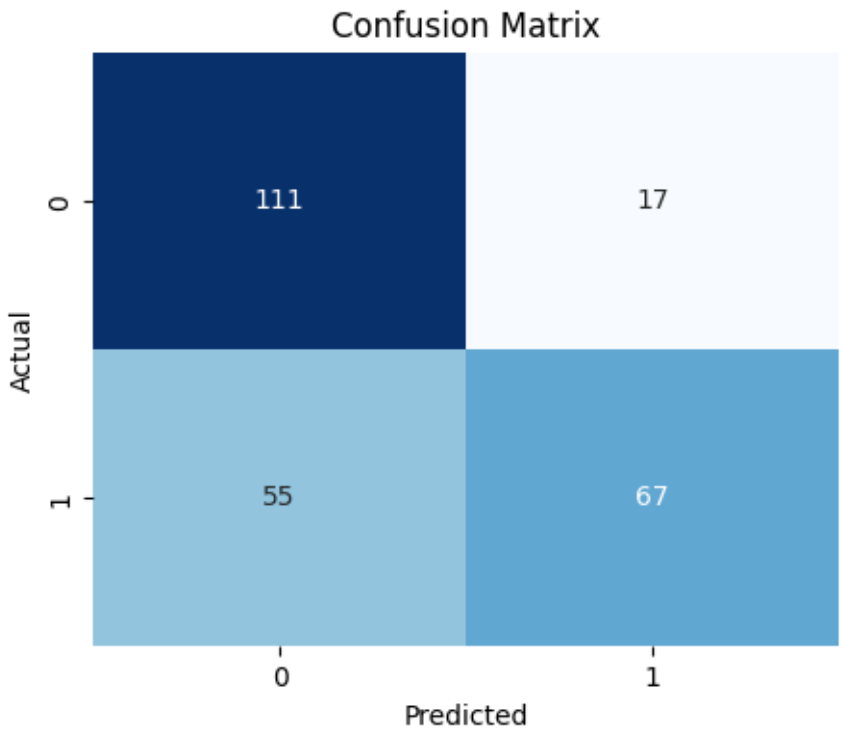
```
# Bag of Words model
cv = CountVectorizer(max_features=1500)
X = cv.fit_transform(corpus).toarray()
y = dataset.iloc[:, 1].values
```

```
# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42)

# Train Random Forest model
model = RandomForestClassifier(n_estimators=501, criterion='entropy', random_state=42)
model.fit(X_train, y_train)
```

```
RandomForestClassifier(criterion='entropy', n_estimators=501, random_state=42)
```

```
# Predictions
y_pred = model.predict(X_test)

# Evaluation
cm = confusion_matrix(y_test, y_pred)
acc = accuracy_score(y_test, y_pred)

print("Confusion Matrix:\n", cm)
print("\nAccuracy:", round(acc * 100, 2), "%")
print("\nClassification Report:\n", classification_report(y_test, y_pred))

plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```

```
Confusion Matrix:
 [[111  17]
 [ 55  67]]

Accuracy: 71.2 %

Classification Report:
              precision    recall  f1-score   support

           0       0.67      0.87      0.76       128
           1       0.80      0.55      0.65       122

    accuracy                           0.71       250
   macro avg       0.73      0.71      0.70       250
weighted avg       0.73      0.71      0.70       250
```



LEARNING OUTCOME:

EXPERIMENT – 12

AIM: Detecting Spam Emails Using Tensorflow in Python

THEORY:

Spam detection with TensorFlow treats emails as sequences and learns a binary classifier that distinguishes spam from ham through embeddings and recurrent modeling, supported by careful balancing, cleaning, and regularized training. The pipeline emphasizes robust preprocessing, sequence preparation, and disciplined evaluation to avoid bias and overfitting.

Data handling

- Labeled emails are loaded from CSV and inspected to reveal ham–spam imbalance; a balanced subset is created by downsampling the majority class for fair training and metrics.
- While downsampling simplifies learning, it may discard useful ham examples; alternatives include oversampling or synthetic methods in broader literature on imbalance handling.

Cleaning and exploration

- Preprocessing removes “Subject”, punctuation, and stopwords, optionally adding stemming/lemmatization to standardize tokens for modeling.
- Word clouds on ham vs spam help visualize frequent discriminative terms and guide feature considerations like vocabulary size and sequence length.

Sequences and labels

- A Keras Tokenizer maps words to integers, texts are transformed to sequences and padded/truncated to a fixed length (e.g., 100) to form uniform tensors.
- Labels are binarized to {0,1}, aligning with sigmoid outputs and binary cross-entropy loss for optimization.

Model design

- A compact Sequential model uses an Embedding layer, a small LSTM to capture order dependencies, a dense ReLU layer, and a sigmoid output for spam probability.

- Training uses Adam with BinaryCrossentropy and stabilization via EarlyStopping and ReduceLROnPlateau callbacks to curb overfitting and adapt learning rates.

Training and metrics

- Validation monitoring with early stopping selects the best epoch; learning-rate scheduling nudges convergence when validation loss plateaus.
- Reported results show about 97%test accuracy on the balanced split, with accuracy curves plotted over epochs; in practice, add precision, recall, and F1 for a fuller picture on imbalance.

SOURCE CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

import string
import nltk
from nltk.corpus import stopwords
from wordcloud import WordCloud
nltk.download('stopwords')

import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from sklearn.model_selection import train_test_split
from keras.callbacks import EarlyStopping, ReduceLROnPlateau

import warnings
warnings.filterwarnings('ignore')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

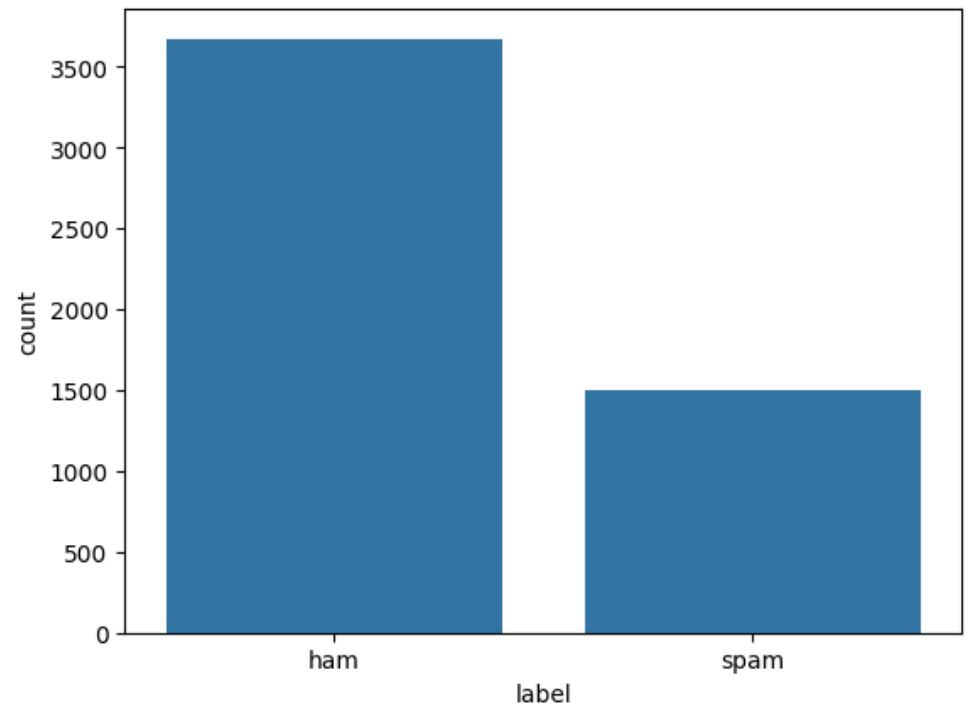
```
data = pd.read_csv('Emails.csv')
data.head()
```

	Unnamed: 0	label	text	label_num
0	605	ham	Subject: enron methanol ; meter # : 988291\r\n...	0
1	2349	ham	Subject: hpl nom for january 9 , 2001\r\n(see...	0
2	3624	ham	Subject: neon retreat\r\nho ho ho , we ' re ar...	0
3	4685	spam	Subject: photoshop , windows , office . cheap ...	1
4	2030	ham	Subject: re : indian springs\r\nthis deal is t...	0

```
data.shape
```

```
(5171, 4)
```

```
sns.countplot(x='label', data=data)
plt.show()
```




```
ham_msg = data[data['label'] == 'ham']
spam_msg = data[data['label'] == 'spam']

# Downsample Ham emails to match the number of Spam emails
ham_msg_balanced = ham_msg.sample(n=len(spam_msg), random_state=42)

# Combine balanced data
balanced_data = pd.concat([ham_msg_balanced, spam_msg]).reset_index(drop=True)

# Visualize the balanced dataset
sns.countplot(x='label', data=balanced_data)
plt.title("Balanced Distribution of Spam and Ham Emails")
plt.xticks(ticks=[0, 1], labels=['Ham (Not Spam)', 'Spam'])
plt.show()
```

```
balanced_data['text'] = balanced_data['text'].str.replace('Subject', '')
balanced_data.head()
```

	Unnamed: 0	label	text	label_num
0	3444	ham	: conoco - big cowboy\r\ndarren :\r\ni ' m not...	0
1	2982	ham	: feb 01 prod : sale to teco gas processing\r\...	0
2	2711	ham	: california energy crisis\r\ncalifornia □ , s...	0
3	3116	ham	: re : nom / actual volume for april 23 rd\r\n...	0
4	1314	ham	: eastrans nomination changes effective 8 / 2 ...	0

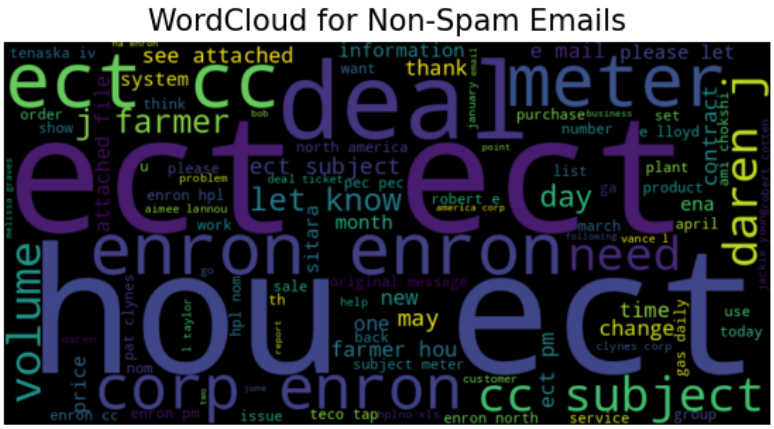
```
punctuations_list = string.punctuation
def remove_punctuations(text):
    temp = str.maketrans('', '', punctuations_list)
    return text.translate(temp)

balanced_data['text']= balanced_data['text'].apply(lambda x: remove_punctuations(x))
balanced_data.head()
```

	Unnamed: 0	label	text	label_num
0	3444	ham	conoco big cowboy\r\ndarren \r\ni m not sur...	0
1	2982	ham	feb 01 prod sale to teco gas processing\r\ns...	0
2	2711	ham	california energy crisis\r\ncalifornia □ s p...	0
3	3116	ham	re nom actual volume for april 23 rd\r\nwe ...	0
4	1314	ham	eastrans nomination changes effective 8 2 0...	0

```
def remove_stopwords(text):
    stop_words = stopwords.words('english')

    imp_words = []

    # Storing the important words
    for word in str(text).split():
        word = word.lower()

        if word not in stop_words:
            imp_words.append(word)

    output = " ".join(imp_words)

    return output


balanced_data['text'] = balanced_data['text'].apply(lambda text: remove_stopwords(text))
balanced_data.head()
```

	Unnamed: 0	label	text	label_num
0	3444	ham	conoco big cowboy darren sure help know else a...	0
1	2982	ham	feb 01 prod sale teco gas processing sale deal...	0
2	2711	ham	california energy crisis california □ power cr...	0
3	3116	ham	nom actual volume april 23 rd agree eileen pon...	0
4	1314	ham	eastrans nomination changes effective 8 2 00 p...	0

```
def plot_word_cloud(data, typ):
    email_corpus = " ".join(data['text'])
    wc = WordCloud(background_color='black', max_words=100, width=800, height=400).generate(email_corpus)
    plt.figure(figsize=(7, 7))
    plt.imshow(wc, interpolation='bilinear')
    plt.title(f'WordCloud for {typ} Emails', fontsize=15)
    plt.axis('off')
    plt.show()

plot_word_cloud(balanced_data[balanced_data['label'] == 'ham'], typ='Non-Spam')
plot_word_cloud(balanced_data[balanced_data['label'] == 'spam'], typ='Spam')
```



```
train_X, test_X, train_Y, test_Y = train_test_split(
    balanced_data['text'], balanced_data['label'], test_size=0.2, random_state=42
)

tokenizer = Tokenizer()
tokenizer.fit_on_texts(train_X)

train_sequences = tokenizer.texts_to_sequences(train_X)
test_sequences = tokenizer.texts_to_sequences(test_X)

max_len = 100  # Maximum sequence length
train_sequences = pad_sequences(train_sequences, maxlen=max_len, padding='post', truncating='post')
test_sequences = pad_sequences(test_sequences, maxlen=max_len, padding='post', truncating='post')

train_Y = (train_Y == 'spam').astype(int)
test_Y = (test_Y == 'spam').astype(int)
```

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Embedding(input_dim=len(tokenizer.word_index) + 1, output_dim=32, input_length=max_len),
    tf.keras.layers.LSTM(16),
    tf.keras.layers.Dense(32, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')  # Output layer
])

model.compile(
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
    optimizer='adam',
    metrics=['accuracy']
)

model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	0 (unbuilt)
lstm (LSTM)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

```
Total params: 0 (0.00 B)
Trainable params: 0 (0.00 B)
Non-trainable params: 0 (0.00 B)
```

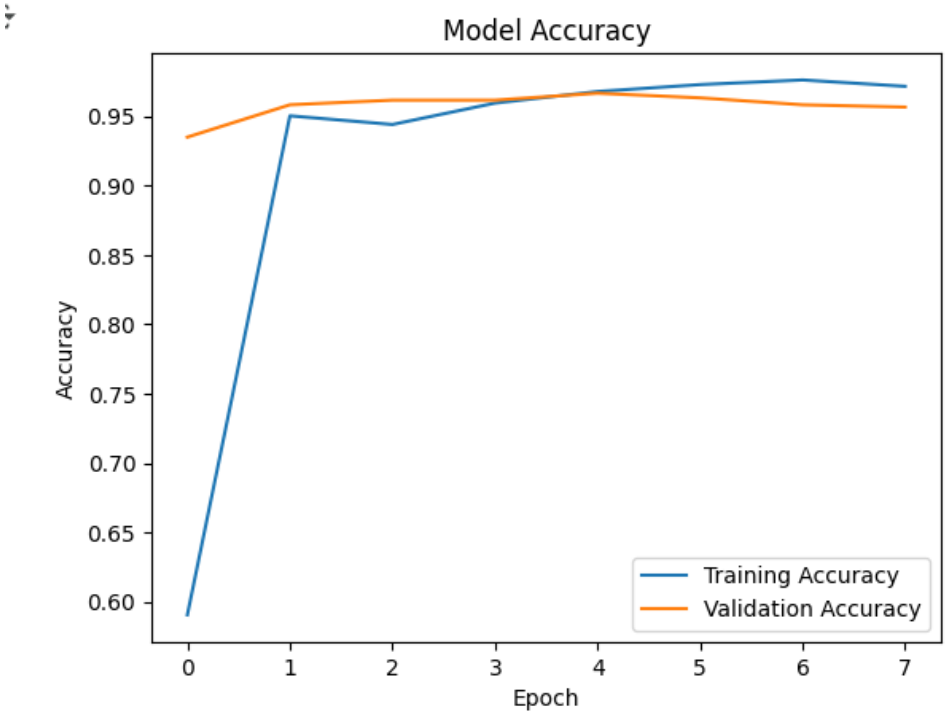
```
es = EarlyStopping(patience=3, monitor='val_accuracy', restore_best_weights=True)
lr = ReduceLROnPlateau(patience=2, monitor='val_loss', factor=0.5, verbose=0)

history = model.fit(
    train_sequences, train_Y,
    validation_data=(test_sequences, test_Y),
    epochs=20,
    batch_size=32,
    callbacks=[lr, es]
)
```

```
Epoch 1/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 13s 127ms/step - accuracy: 0.5425 - loss: 0.6892 - val_accuracy: 0.9350 - val_loss: 0.3689 - learning_rate: 0.0010
Epoch 2/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 5s 56ms/step - accuracy: 0.9489 - loss: 0.2763 - val_accuracy: 0.9583 - val_loss: 0.1530 - learning_rate: 0.0010
Epoch 3/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 4s 45ms/step - accuracy: 0.9523 - loss: 0.2029 - val_accuracy: 0.9617 - val_loss: 0.1809 - learning_rate: 0.0010
Epoch 4/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 7s 75ms/step - accuracy: 0.9577 - loss: 0.1826 - val_accuracy: 0.9617 - val_loss: 0.1508 - learning_rate: 0.0010
Epoch 5/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 8s 49ms/step - accuracy: 0.9688 - loss: 0.1320 - val_accuracy: 0.9667 - val_loss: 0.1388 - learning_rate: 0.0010
Epoch 6/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 5s 45ms/step - accuracy: 0.9712 - loss: 0.1285 - val_accuracy: 0.9633 - val_loss: 0.1538 - learning_rate: 0.0010
Epoch 7/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 4s 47ms/step - accuracy: 0.9797 - loss: 0.0992 - val_accuracy: 0.9583 - val_loss: 0.1751 - learning_rate: 0.0010
Epoch 8/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step - accuracy: 0.9726 - loss: 0.1253 - val_accuracy: 0.9567 - val_loss: 0.1783 - learning_rate: 5.0000e-04
```

```
test_loss, test_accuracy = model.evaluate(test_sequences, test_Y)
print('Test Loss :',test_loss)
print('Test Accuracy :',test_accuracy)
```

```
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step - accuracy: 0.9730 - loss: 0.1173
Test Loss : 0.1388220489025116
Test Accuracy : 0.9666666388511658
```

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



LEARNING OUTCOME: