योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION

VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution

SCHOOL OF ENGINEERING & TECHNOLOGY

B. Tech Programme: AI-ML (A)

(5th Semester)

Course Title: Fundamentals of Deep Learning Lab

Course Code: AIML- 355

Submitted To:

Ms. Prabhneet Kaur

Submitted By:

Name: Kunsh Sabharwal

Enrolment No: 01117711623

VIPS
योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION
VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

VISION OF INSTITUTE

To be an educational institute that empowers the field of engineering to build a sustainable future by providing quality education with innovative practices that supports people, planet and profit.

MISSION OF INSTITUTE

To groom the future engineers by providing value-based education and awakening students' curiosity, nurturing creativity and building capabilities to enable them to make significant contributions to the world.

योगः कर्मसु कौशलम्
IN PURSUIT OF PERFECTION
VIVEKANANDA INSTITUTE OF PROFESSIONAL STUDIES - TECHNICAL CAMPUS
Grade A++ Accredited Institution by NAAC
NBA Accredited for MCA Programme; Recognized under Section 2(f) by UGC;
Affiliated to GGSIP University, Delhi; Recognized by Bar Council of India and AICTE
An ISO 9001:2015 Certified Institution
SCHOOL OF ENGINEERING & TECHNOLOGY

INDEX

S. No.	Experiment	Date	Marks			Remarks	Updated Marks	Faculty Signature
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			
1.								
2.								
3.								
4.								

5.								
6.								
7.								
8.								
9.								
10.								

11.								
12.								
13.								

EXPERIMENT 1

Aim: To explore the basic features of Tensorflow and Keras packages in python

Objectives:

1) Understand the foundational functionalities of Tensorflow and Keras for deep learning
2) Explore how to build, compile and train basic neural network models using Keras.

Theory:

1. Tensorflow:

Definition:

- TensorFlow is an open-source deep learning framework developed by Google for building and training machine learning models.
- It represents data as tensors and computations as graphs, enabling efficient execution on CPUs, GPUs, and TPUs.
- With its high-level (tf.keras) API, TensorFlow simplifies creating, compiling, and deploying neural networks.

Key features:

- **Tensor-Based Computation:** Uses tensors (multi-dimensional arrays) for efficient data representation and processing. Optimized for CPU, GPU, and TPU execution.
- **Eager Execution:** Runs operations instantly for easier debugging. Makes code more intuitive and Python-like.
- **Integrated Keras API:** Simplifies neural network creation and training. Offers high-level yet flexible model building.
- **Automatic Differentiation:** Computes gradients automatically for training. Supports custom training loops.
- **Deployment Versatility:** Deploy models to servers, mobile, web, or embedded devices. Supports TensorFlow Serving, Lite, and.js.

Core concepts:

- **Tensors:** Fundamental data structure in TensorFlow; multi-dimensional arrays used to store and manipulate data.
- **Computation Graph:** Represents operations as nodes and data (tensors) as edges; defines the flow of computation.
- **Automatic Differentiation:** Uses backpropagation to compute gradients automatically for training deep learning models.

Case studies:

1. **Google Translate – Neural Machine Translation:**

Google uses TensorFlow to power its Neural Machine Translation (NMT) system, which translates text between 100+ languages in Google Translate. TensorFlow's GPU acceleration

and sequence-to-sequence models enable real-time, high-quality translations for billions of users daily.

2. **Tesla – Autopilot Computer Vision:**

Tesla applies TensorFlow in its Autopilot system for real-time object detection and lane recognition from camera feeds. The deep learning models process visual data on embedded hardware to assist in navigation, obstacle avoidance, and driving safety.

2. Keras

Definition:

- Keras is an open-source, high-level deep learning API written in Python that runs on top of frameworks like TensorFlow.
- It simplifies building, training, and evaluating neural networks with a user-friendly and modular interface, making it ideal for beginners and rapid prototyping.

Key features:

- **User-Friendly API:** Simple, intuitive, and readable syntax for building models quickly.
- **Modular Design:** Models are built by combining independent, reusable components like layers, optimizers, and loss functions.
- **Multiple Backend Support:** Can run on TensorFlow, Theano, or CNTK (with TensorFlow as the default).
- **Pretrained Models:** Includes popular pretrained architectures like VGG, ResNet, and Inception for transfer learning.
- **Seamless GPU/CPU Switching:** Automatically utilizes available hardware without changing the code.

Core concepts:

- **Models:** Central structure in Keras, either Sequential (layer-by-layer) or Functional (complex architectures).
- **Layers:** Building blocks of neural networks, such as Dense, Convolutional, and Recurrent layers.
- **Compilation & Training:** Models are compiled with optimizers, loss functions, and metrics, then trained on data using .fit().

Case studies:

1. **NASA – Solar Flare Prediction:**

NASA researchers have used Keras with TensorFlow backend to build deep learning models that predict solar flares from satellite data. The models analyse time-series and image data to forecast space weather events that can impact satellites and communications.

2. **Netflix – Personalized Content Recommendations**:

Netflix has leveraged Keras for rapid prototyping of deep learning models that recommend personalized shows and movies. By analysing user viewing patterns, Keras-based models help enhance user engagement and retention.

Regression and classification with Tensorflow and Keras:

Regression using TensorFlow & Keras:

Regression predicts continuous numerical outputs by mapping input features to a single value. In TensorFlow with Keras, layers process data through weighted connections, applying activation functions. The model minimizes loss (e.g., MSE) using optimizers like Adam, learning patterns to make accurate continuous predictions.

Classification using TensorFlow & Keras:

Classification assigns input data to discrete categories. In TensorFlow with Keras, neural network layers extract features, and the final layer uses sigmoid (binary) or softmax (multi-class) activation. The model minimizes classification loss (e.g., cross-entropy) to learn decision boundaries, enabling accurate label predictions for unseen data.

Code using TensorFlow and Keras for regression (House Price Prediction):

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

housing = fetch_california_housing()
X, y = housing.data, housing.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    layers.Dense(32, activation='relu'),
    layers.Dense(1)
])

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=1)

mse, mae = model.evaluate(X_test, y_test, verbose=0)
print(f"Mean Squared Error: {mse:.4f}")
print(f"Mean Absolute Error: {mae:.4f}")

dummy_data = np.array([[8.3252, 41.0, 6.9841, 1.0238, 322.0, 2.5556, 37.88, -122.23]])
dummy_data = scaler.transform(dummy_data)
predicted_price = model.predict(dummy_data)[0][0]

print(f"Predicted Median House Price: ${predicted_price * 100000:.2f}")
```

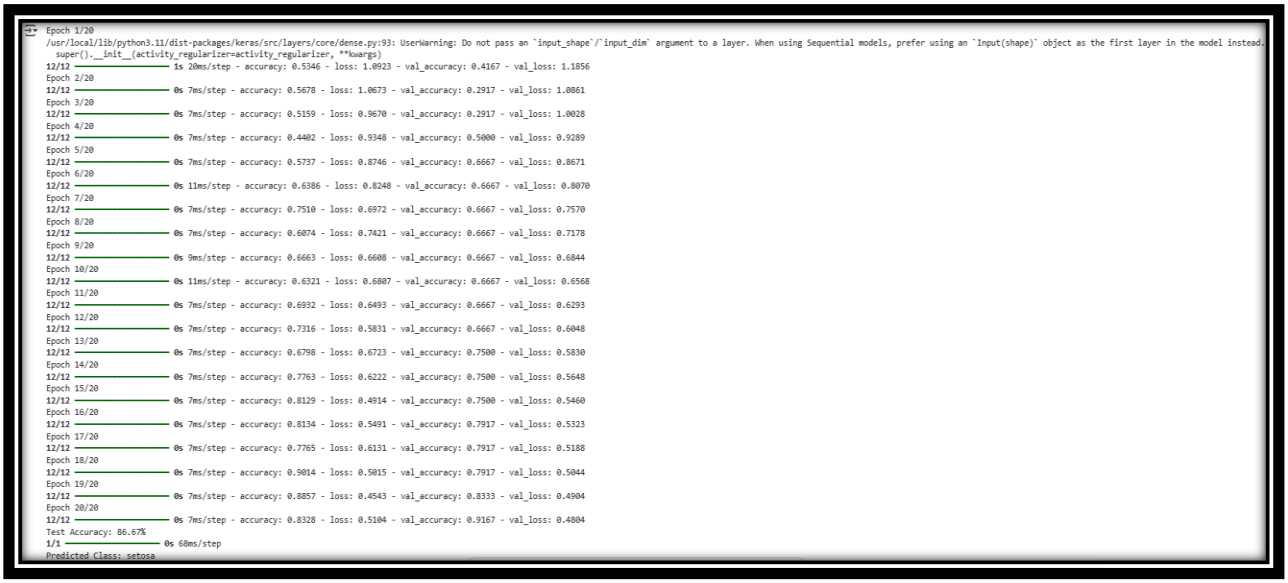
Output:

```
Epoch 1/10
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 1.7439 - mae: 0.9375 - val_loss: 0.4856 - val_mae: 0.4799
Epoch 2/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.4282 - mae: 0.4672 - val_loss: 0.4358 - val_mae: 0.4583
Epoch 3/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.4097 - mae: 0.4384 - val_loss: 0.3977 - val_mae: 0.4486
Epoch 4/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - loss: 0.3505 - mae: 0.4236 - val_loss: 0.3848 - val_mae: 0.4326
Epoch 5/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3566 - mae: 0.4214 - val_loss: 0.3713 - val_mae: 0.4223
Epoch 6/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3403 - mae: 0.4102 - val_loss: 0.3658 - val_mae: 0.4434
Epoch 7/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3347 - mae: 0.4105 - val_loss: 0.3683 - val_mae: 0.4119
Epoch 8/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3289 - mae: 0.3991 - val_loss: 0.3484 - val_mae: 0.4183
Epoch 9/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - loss: 0.3102 - mae: 0.3880 - val_loss: 0.3495 - val_mae: 0.4059
Epoch 10/10
413/413 ━━━━━━━━━━━━━━━━━━━━ 2s 3ms/step - loss: 0.3134 - mae: 0.3887 - val_loss: 0.3392 - val_mae: 0.4067
Mean Squared Error: 0.3305
Mean Absolute Error: 0.3999
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 63ms/step
Predicted Median House Price: $431787.91
```

Code using Tensorflow and Keras for classification (Iris dataset):

```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

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

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

model = keras.Sequential([
    layers.Dense(16, activation='relu', input_shape=(X_train.shape[1],)),
    layers.Dense(8, activation='relu'),
    layers.Dense(3, activation='softmax')  # 3 classes in Iris dataset
])

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

model.fit(X_train, y_train, epochs=20, batch_size=8, validation_split=0.2, verbose=1)

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

dummy_flower = np.array([[5.1, 3.5, 1.4, 0.2]])
dummy_flower = scaler.transform(dummy_flower)

prediction = model.predict(dummy_flower)
predicted_class = np.argmax(prediction)

print(f"Predicted Class: {iris.target_names[predicted_class]}")
```

Output:



```
Epoch 1/20
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
12/12 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.5346 - loss: 1.0923 - val_accuracy: 0.4167 - val_loss: 1.1856
Epoch 2/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5678 - loss: 1.0673 - val_accuracy: 0.2917 - val_loss: 1.0861
Epoch 3/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5159 - loss: 0.9670 - val_accuracy: 0.2917 - val_loss: 1.0028
Epoch 4/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.4402 - loss: 0.9348 - val_accuracy: 0.5000 - val_loss: 0.9289
Epoch 5/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.5737 - loss: 0.8746 - val_accuracy: 0.6667 - val_loss: 0.8671
Epoch 6/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6386 - loss: 0.8248 - val_accuracy: 0.6667 - val_loss: 0.8070
Epoch 7/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7510 - loss: 0.6972 - val_accuracy: 0.6667 - val_loss: 0.7570
Epoch 8/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6074 - loss: 0.7421 - val_accuracy: 0.6667 - val_loss: 0.7178
Epoch 9/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.6663 - loss: 0.6608 - val_accuracy: 0.6667 - val_loss: 0.6844
Epoch 10/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.6321 - loss: 0.6807 - val_accuracy: 0.6667 - val_loss: 0.6568
Epoch 11/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6932 - loss: 0.6493 - val_accuracy: 0.6667 - val_loss: 0.6293
Epoch 12/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7316 - loss: 0.5831 - val_accuracy: 0.6667 - val_loss: 0.6048
Epoch 13/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.6798 - loss: 0.6723 - val_accuracy: 0.7500 - val_loss: 0.5830
Epoch 14/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7763 - loss: 0.6222 - val_accuracy: 0.7500 - val_loss: 0.5648
Epoch 15/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8129 - loss: 0.4914 - val_accuracy: 0.7500 - val_loss: 0.5460
Epoch 16/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8134 - loss: 0.5491 - val_accuracy: 0.7917 - val_loss: 0.5323
Epoch 17/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.7765 - loss: 0.6131 - val_accuracy: 0.7917 - val_loss: 0.5188
Epoch 18/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9014 - loss: 0.5015 - val_accuracy: 0.7917 - val_loss: 0.5044
Epoch 19/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8857 - loss: 0.4543 - val_accuracy: 0.8333 - val_loss: 0.4904
Epoch 20/20
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.8328 - loss: 0.5104 - val_accuracy: 0.9167 - val_loss: 0.4804
Test Accuracy: 86.67%
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 68ms/step
Predicted Class: setosa
```

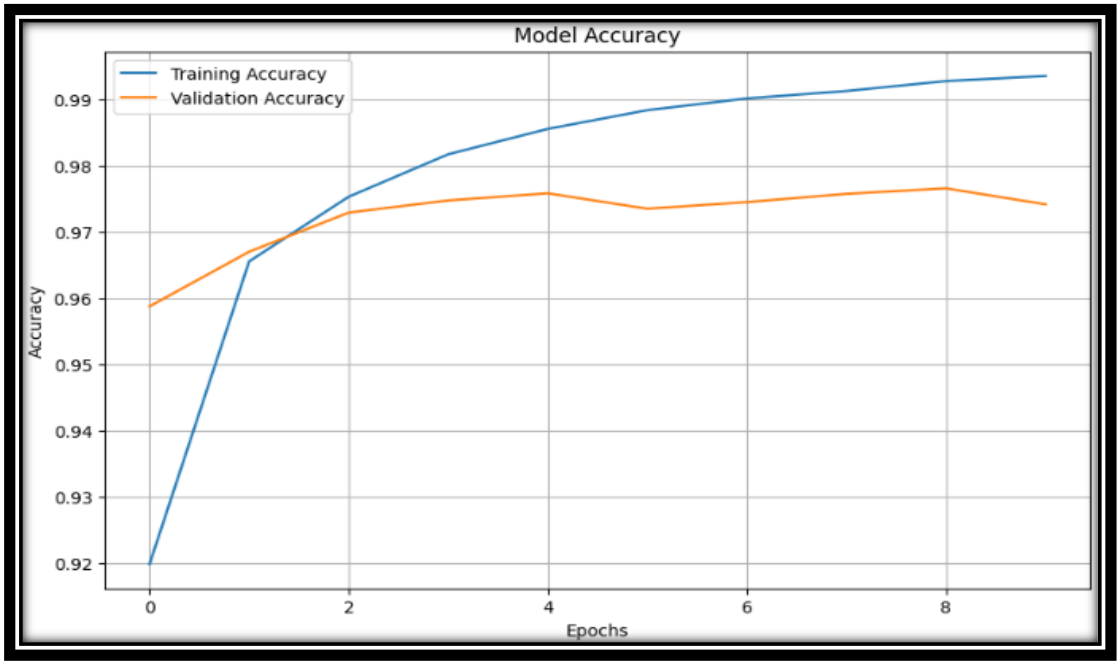
Code for MNIST Dataset classification using Tensorflow and Keras:

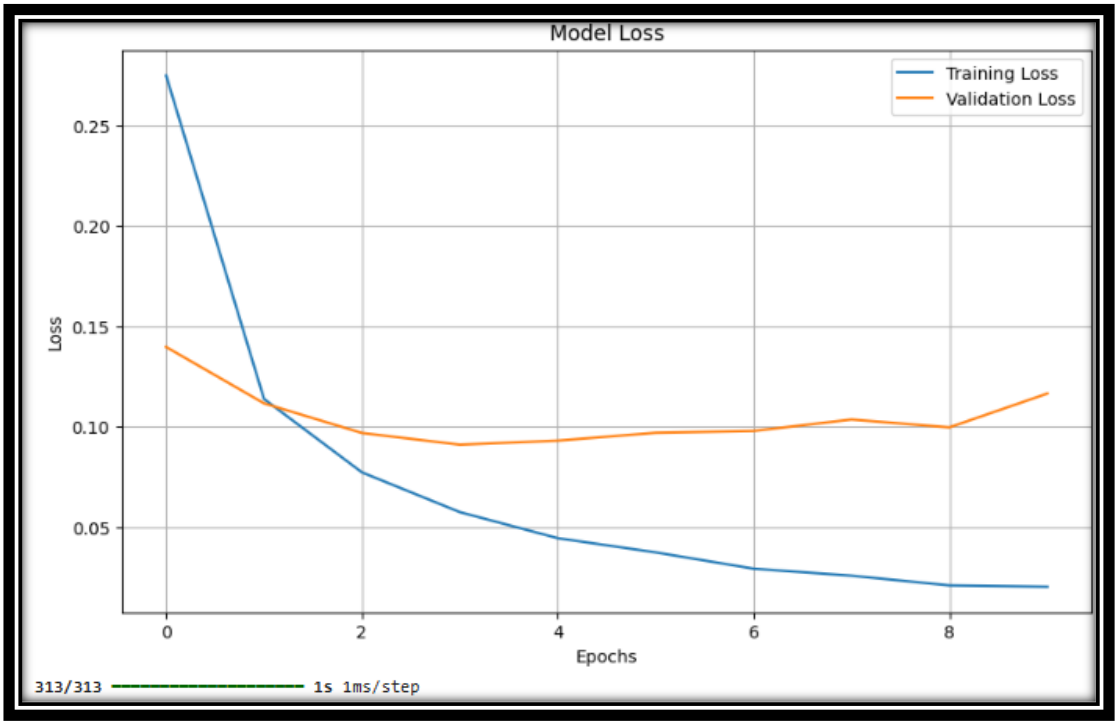
```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt

# Load the MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Normalize the data
X_train = X_train.astype('float32') / 255.0
X_test = X_test.astype('float32') / 255.0

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

# Build the ANN model
model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the 28x28 images into a 1D vector
    Dense(128, activation='relu'),  # Hidden layer with 128 neurons
    Dense(64, activation='relu'),   # Hidden layer with 64 neurons
    Dense(10, activation='softmax') # Output layer for 10 classes
])

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

# Train the model
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy:.2f}")

# Plot training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
```

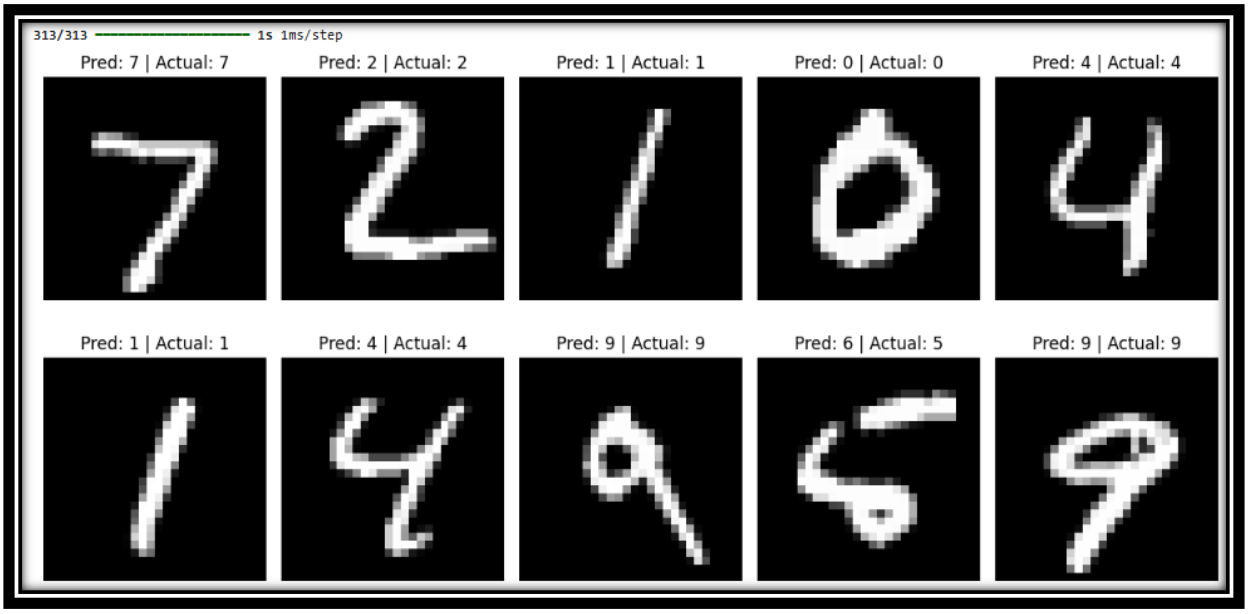
```
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.2, verbose=1)

# Evaluate the model on the test data
test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)
print(f"Test Accuracy: {test_accuracy:.2f}")

# Plot training and validation accuracy
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.grid()
plt.show()

# 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 Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()

# Predict on test data and visualize some predictions
import numpy as np
predictions = np.argmax(model.predict(X_test), axis=1)
actual = np.argmax(y_test, axis=1)

# Visualize first 10 test samples
plt.figure(figsize=(12, 6))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.imshow(X_test[i], cmap='gray')
    plt.title(f"Pred: {predictions[i]} | Actual: {actual[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
/usr/local/lib/python3.11/dist-packages/keras/src/layers/reshaping/flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.8597 - loss: 0.4875 - val_accuracy: 0.9588 - val_loss: 0.1397
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9626 - loss: 0.1223 - val_accuracy: 0.9670 - val_loss: 0.1117
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 4ms/step - accuracy: 0.9743 - loss: 0.0809 - val_accuracy: 0.9729 - val_loss: 0.0969
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9825 - loss: 0.0565 - val_accuracy: 0.9747 - val_loss: 0.0911
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9865 - loss: 0.0420 - val_accuracy: 0.9758 - val_loss: 0.0931
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 8s 5ms/step - accuracy: 0.9900 - loss: 0.0327 - val_accuracy: 0.9735 - val_loss: 0.0970
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.9916 - loss: 0.0250 - val_accuracy: 0.9745 - val_loss: 0.0979
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 7s 5ms/step - accuracy: 0.9924 - loss: 0.0221 - val_accuracy: 0.9758 - val_loss: 0.1036
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.9932 - loss: 0.0202 - val_accuracy: 0.9766 - val_loss: 0.0997
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 4ms/step - accuracy: 0.9944 - loss: 0.0166 - val_accuracy: 0.9742 - val_loss: 0.1166
Test Accuracy: 0.98
```







Learning Outcome:

EXPERIMENT 2

Aim: Implementation of ANN model for regression and classification problems in Python

Objectives:

1) To understand and implement the fundamentals of Artificial Neural Network
2) Implement an ANN for regression task
3) Implement an ANN for classification task
4) Evaluate and compare model performances

Theory: Artificial Neural Networks (ANNs) are computational models inspired by the human brain, used to approximate complex functions. ANNs consist of layers: input layers receive data, hidden layers extract features through weighted connections and activations, and the output layer produces predictions for tasks like classification or regression.

Input Layer:

This is the entry point of the network where the raw input data (features) is fed in. It does not perform any computations but simply passes the data to the next layer.

Hidden Layers:

These are the core computational layers of the network. Each hidden layer consists of neurons that apply weighted sums and biases to inputs, followed by an activation function to introduce non-linearity. Multiple hidden layers allow the network to learn complex patterns and representations from the data.

Output Layer:

The last layer produces the final output. For regression tasks, it usually has one neuron with a linear activation, while for classification tasks, it uses activation functions like softmax or sigmoid to output probabilities or class scores.

Parts of ANN:

1) **Neurons (Nodes):**
   The fundamental processing units that receive inputs, apply weights and biases, compute a weighted sum, pass the result through an activation function, and send the output to the next layer.
2) **Layers:**
   Organized groups of neurons. There are three main types:
   (a) **Input layer:** receives the raw data features.
   (b) **Hidden layers:** perform feature extraction and complex transformations. Multiple hidden layers enable deep learning.
   (c) **Output layer:** produces the final prediction or classification result.
3) **Weights:**
   Numeric values assigned to connections between neurons. They control how much influence an input has on the neuron's output and are adjusted during training.
4) **Bias:**
   An extra adjustable parameter added to the weighted sum before applying the activation function. It allows the network to better fit the data by shifting the activation function.
5) **Activation Functions:**
   Non-linear functions like ReLU, Sigmoid, or Tanh applied to neuron outputs. They

introduce non-linearity, enabling the network to model complex relationships beyond linear combinations.

6) **Input:**
The initial data features fed into the network, typically normalized or scaled for better training performance.

7) **Output:**
The final layer's response, which can be continuous values (for regression) or class probabilities/scores (for classification).

Regression: It is a machine learning task where the goal is to predict a continuous numerical value based on input features. It models the relationship between dependent and independent variables to estimate or forecast outcomes, commonly used in applications like housing price prediction, stock forecasting, and more.

Classification: It is a supervised machine learning task where the goal is to assign input data into one of several predefined categories or classes. It learns patterns from labelled data to predict discrete labels, commonly used in tasks like spam detection, image recognition, and medical diagnosis.

Binary Classification: Categorizes data into two classes using a sigmoid-activated output neuron.
Multiclass Classification: Assigns inputs to one of multiple classes using one output neuron per class.
Softmax Classification: Applies the softmax function to output logits to produce class probabilities that sum to one.

Source Code: (a) Regression on Boston Housing Dataset

```
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
import numpy as np

(x_train, y_train), (x_test, y_test) = tf.keras.datasets.boston_housing.load_data()

scaler = StandardScaler()
x_train = scaler.fit_transform(x_train)
x_test = scaler.transform(x_test)

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(x_train.shape[1],)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(1)
])

model.compile(optimizer='sgd', loss='mse', metrics=['mae'])

model.fit(x_train, y_train, epochs=10)

loss, mae = model.evaluate(x_test, y_test, verbose=0)
print(f"Test set Mean Absolute Error: {mae:.2f}")
print(f"Test set Loss (MSE): {loss:.2f}")
```

(b) Classification on MNIST Dataset

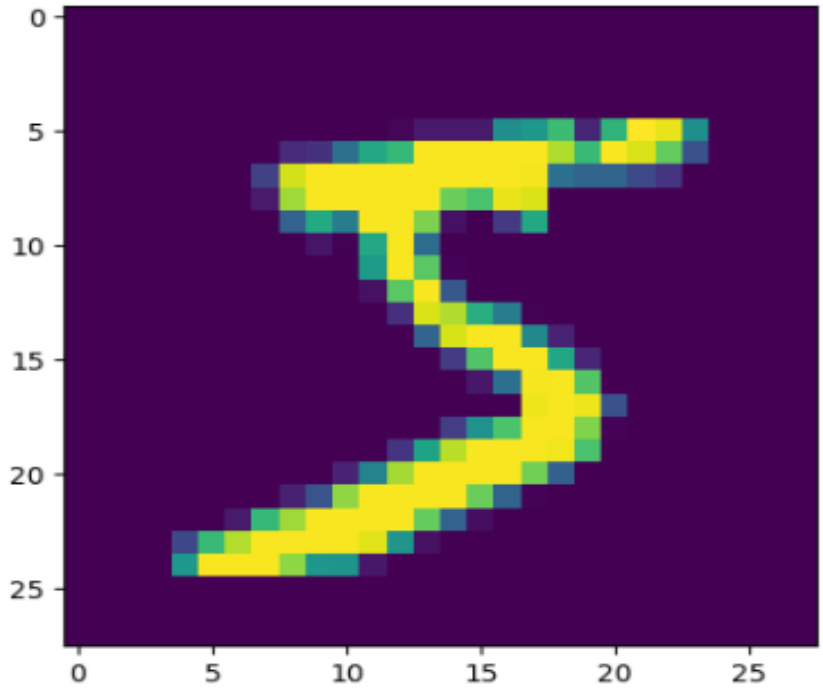
```
import tensorflow as tf
from keras.layers import Dense, Flatten, Input
mnist=tf.keras.datasets.mnist
(x_train, y_train), (x_test, y_test)=mnist.load_data()
x_train.shape, y_train.shape, x_test.shape, y_test.shape
import matplotlib.pyplot as plt
plt.imshow(x_train[0])
model=tf.keras.Sequential([
    Input((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=10, batch_size=100, verbose=1)
loss, acc = model.evaluate(x_test, y_test)
print(f"Test Loss: {loss:.4f}")
print(f"Test Accuracy: {acc:.4f}")
import numpy as np
pred = model.predict(x_train[6].reshape(1,28,28))
np.argmax(pred)
```

Output: (a) Regression on Boston Housing:

```
Epoch 1/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 282.7880 - mae: 13.4613
Epoch 2/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 16.2076 - mae: 3.0239
Epoch 3/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 19.9333 - mae: 3.0783
Epoch 4/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.3263 - mae: 2.4173
Epoch 5/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 11.4513 - mae: 2.3452
Epoch 6/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 10.4147 - mae: 2.4087
Epoch 7/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 10.3718 - mae: 2.3546
Epoch 8/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.8920 - mae: 2.3444
Epoch 9/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9.3307 - mae: 2.2253
Epoch 10/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 12.7421 - mae: 2.4333
Test set Mean Absolute Error: 3.38
Test set Loss (MSE): 24.67
```

(b) Classification on MNIST Dataset:

```
Epoch 1/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.8079 - loss: 5.3928
Epoch 2/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9299 - loss: 0.4594
Epoch 3/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9489 - loss: 0.2615
Epoch 4/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9588 - loss: 0.1824
Epoch 5/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9643 - loss: 0.1418
Epoch 6/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9688 - loss: 0.1275
Epoch 7/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9715 - loss: 0.1124
Epoch 8/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.9755 - loss: 0.0946
Epoch 9/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9765 - loss: 0.0828
Epoch 10/10
600/600 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.9791 - loss: 0.0752
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9542 - loss: 0.2206
Test Loss: 0.2000
Test Accuracy: 0.9595
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 60ms/step
np.int64(1)
```



Learning Outcome:

EXPERIMENT 3

Aim: Implementation of Convolutional Neural Network (CNN) for MRI Dataset in Python

Objectives:

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

Theory:

Convolutional Neural Networks (CNNs) are a specialized class of deep learning models designed primarily for analysing visual data. They are particularly effective in tasks such as medical imaging, object recognition, and pattern classification because they can automatically learn hierarchical spatial features from raw images without manual feature engineering.

Basics of CNN:

A CNN processes input images through a sequence of layers that progressively extract higher-level features. In the context of MRI data, CNNs are well-suited for identifying subtle patterns, textures, and anomalies that may be critical for diagnosis.

Key Components of CNN:

1. **Convolutional Layer -**
    - Applies learnable filters (kernels) that scan over the image to capture local spatial features such as edges, textures, or shapes.
    - Helps preserve spatial relationships between pixels.
2. **Activation Function (ReLU) -**
    - Introduces non-linearity, ensuring the network can learn complex patterns.
    - Rectified Linear Unit (ReLU) is the most common choice.
3. **Pooling Layer -**
    - Reduces the spatial dimensions of feature maps by summarizing local regions.
    - Max pooling is widely used, which selects the maximum value in each patch, reducing computation and controlling overfitting.
4. **Fully Connected (Dense) Layer -**
    - Flattens the extracted features into a vector and connects them to output neurons.

- This stage integrates all learned features to make the final classification.

5. Output Layer -

- Uses activation functions such as **Softmax** (for multi-class classification) or **Sigmoid** (for binary classification).
- Produces class probabilities or labels for MRI scans (e.g., tumor vs. non-tumor).

Training Process

- The CNN learns by minimizing a **loss function** (e.g., categorical cross-entropy) through **backpropagation** and optimization (commonly using Adam or SGD).
- During training, filters automatically adjust to detect increasingly complex structures within the MRI scans.

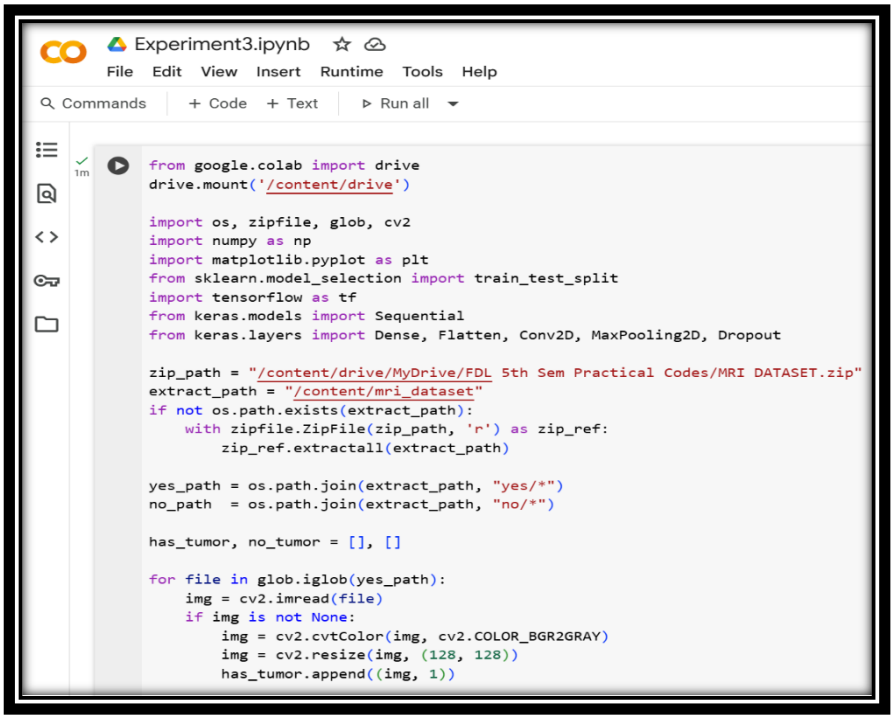
Advantages of CNN in MRI Analysis

1. Automatically extracts hierarchical features without manual intervention.
2. Reduces computational complexity compared to traditional fully connected networks.
3. Highly effective in detecting anomalies, lesions, and patterns in medical imaging.

Conclusion

CNNs form the foundation of modern medical image analysis, offering a powerful approach to detecting and classifying features in MRI scans. Their layered architecture enables them to capture both low-level and high-level image details, making them a reliable tool in healthcare-related deep learning applications.

Source Code:



```
for file in glob.iglob(no_path):
    img = cv2.imread(file)
    if img is not None:
        img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(img, (128, 128))
        no_tumor.append((img, 0))

all_data = has_tumor + no_tumor
np.random.shuffle(all_data)

data   = np.array([item[0] for item in all_data])
labels = np.array([item[1] for item in all_data])

x_train, x_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

x_train = x_train.reshape(x_train.shape[0], 128, 128, 1) / 255.0
x_test  = x_test.reshape(x_test.shape[0], 128, 128, 1) / 255.0

model = Sequential([
    Conv2D(64, kernel_size=3, activation='relu', input_shape=(128, 128, 1)),
    MaxPooling2D(pool_size=(2,2)),
    Conv2D(32, kernel_size=3, activation='relu'),
    MaxPooling2D(pool_size=(2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])
```

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

history = model.fit(x_train, y_train, batch_size=32, epochs=10, validation_data=(x_test, y_test), verbose=1)

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=0)
print(f"\nTest accuracy: {test_acc:.4f}")

plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
plt.plot(history.history['accuracy'], label='Training Acc')
plt.plot(history.history['val_accuracy'], label='Validation Acc')
plt.legend(); plt.title("Model Accuracy")

plt.subplot(1,2,2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend(); plt.title("Model Loss")

plt.show()

predictions = model.predict(x_test)
predicted_classes = (predictions > 0.5).astype(int).flatten()

plt.figure(figsize=(15,10))
for i in range(12):
    plt.subplot(3,4,i+1)
    plt.imshow(x_test[i].reshape(128,128), cmap="gray")
    plt.title(f"True: {y_test[i]}, Pred: {predicted_classes[i]}")
    plt.axis("off")
plt.tight_layout()
plt.show()
```

Outputs:

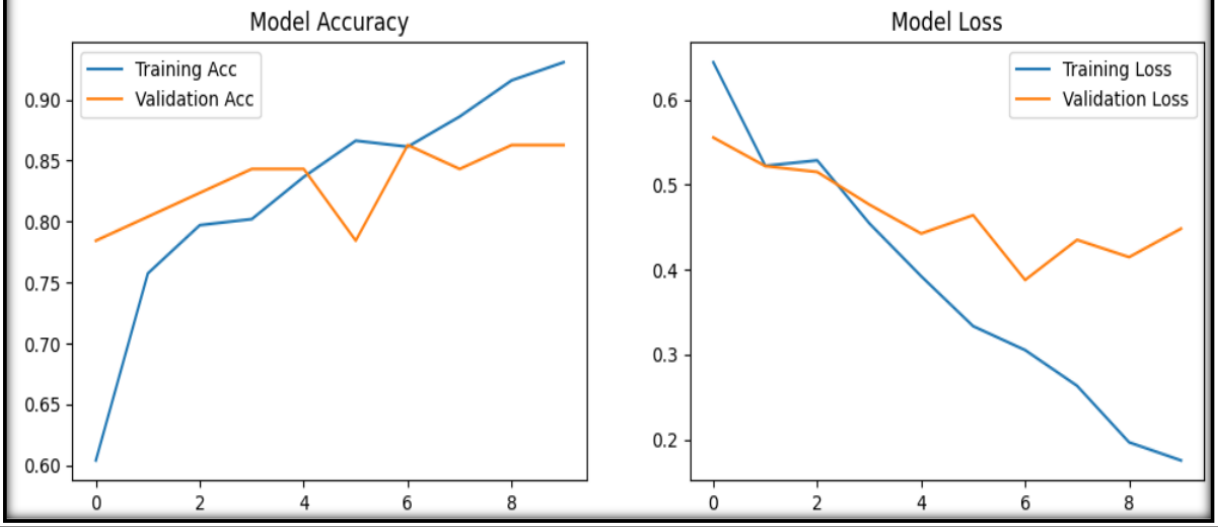
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

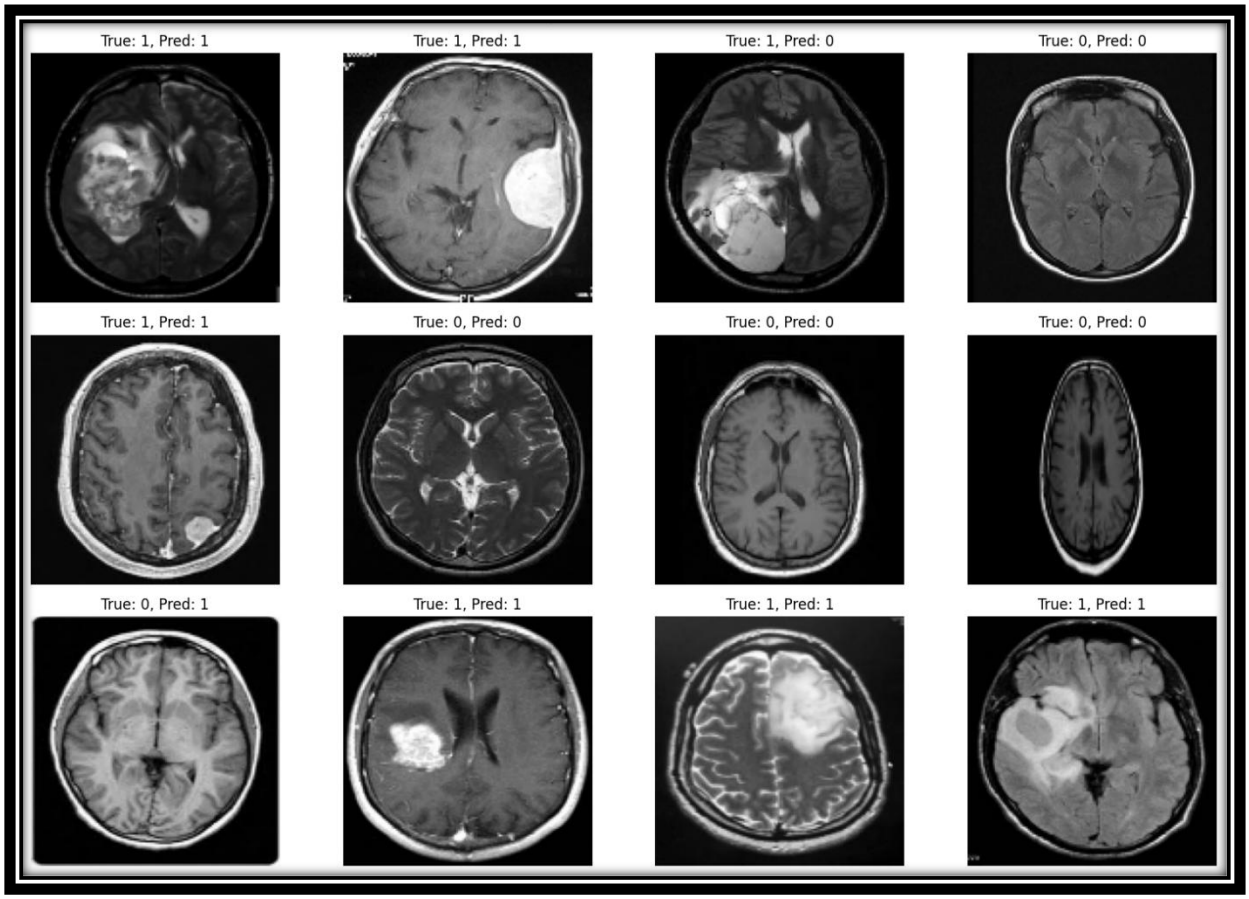
Model: "sequential_2"

Layer (type)	Output Shape	Param #
conv2d_5 (Conv2D)	(None, 126, 126, 64)	640
max_pooling2d_5 (MaxPooling2D)	(None, 63, 63, 64)	0
conv2d_6 (Conv2D)	(None, 61, 61, 32)	18,464
max_pooling2d_6 (MaxPooling2D)	(None, 30, 30, 32)	0
flatten_2 (Flatten)	(None, 28800)	0
dense_5 (Dense)	(None, 128)	3,686,528
dropout_3 (Dropout)	(None, 128)	0
dense_6 (Dense)	(None, 64)	8,256
dropout_4 (Dropout)	(None, 64)	0
dense_7 (Dense)	(None, 1)	65

Total params: 3,713,953 (14.17 MB)
Trainable params: 3,713,953 (14.17 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 959ms/step - accuracy: 0.5611 - loss: 0.6617 - val_accuracy: 0.7843 - val_loss: 0.5554
Epoch 2/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 11s 1s/step - accuracy: 0.7474 - loss: 0.5246 - val_accuracy: 0.8039 - val_loss: 0.5218
Epoch 3/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 1s/step - accuracy: 0.7793 - loss: 0.5219 - val_accuracy: 0.8235 - val_loss: 0.5148
Epoch 4/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 9s 919ms/step - accuracy: 0.8242 - loss: 0.4139 - val_accuracy: 0.8431 - val_loss: 0.4765
Epoch 5/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 11s 900ms/step - accuracy: 0.8262 - loss: 0.4071 - val_accuracy: 0.8431 - val_loss: 0.4424
Epoch 6/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 11s 1s/step - accuracy: 0.8566 - loss: 0.3398 - val_accuracy: 0.7843 - val_loss: 0.4640
Epoch 7/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 9s 1s/step - accuracy: 0.8591 - loss: 0.3145 - val_accuracy: 0.8627 - val_loss: 0.3877
Epoch 8/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 8s 1s/step - accuracy: 0.8978 - loss: 0.2524 - val_accuracy: 0.8431 - val_loss: 0.4350
Epoch 9/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 7s 1s/step - accuracy: 0.9230 - loss: 0.1771 - val_accuracy: 0.8627 - val_loss: 0.4146
Epoch 10/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 10s 953ms/step - accuracy: 0.9502 - loss: 0.1812 - val_accuracy: 0.8627 - val_loss: 0.4481

Test accuracy: 0.8627
```





Learning Outcome:

EXPERIMENT 4

Aim: Implementation of Autoencoders for Dimensionality Reduction in Python.

Objectives:

- Reduce data dimensionality while preserving important features.
- Increase computational efficiency by working with lower-dimensional representation.
- Evaluate reconstruction accuracy to ensure minimal loss of information.

Theory:

Autoencoders:

Autoencoders are a type of unsupervised neural network architecture primarily used for representation learning, dimensionality reduction, and data reconstruction. They are trained to reconstruct their input, effectively learning a compressed, meaningful internal representation (encoding) of the data.

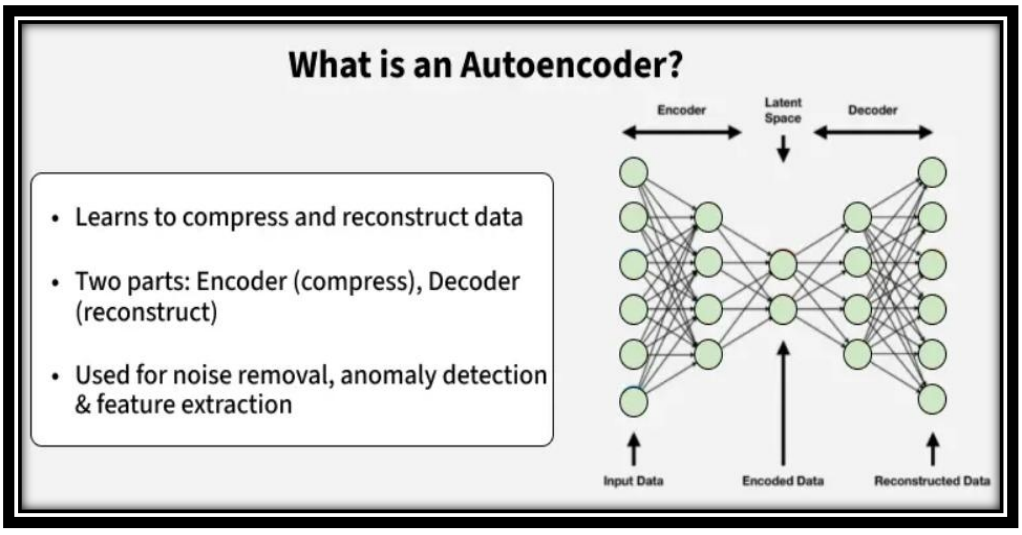
Architecture Overview

An autoencoder consists of two main components:

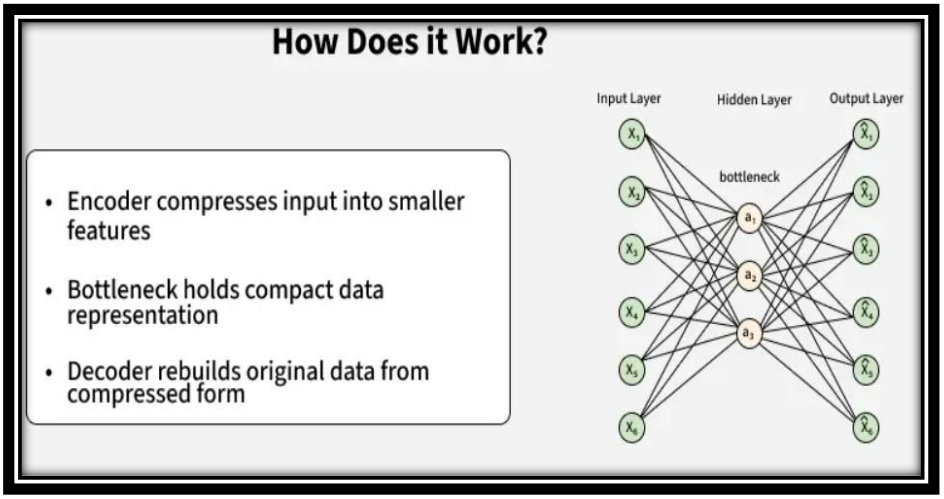
1. **Encoder:** Maps the high-dimensional input data to a lower-dimensional latent space (also called the bottleneck). It consists of one or more dense or recurrent layers, depending on the input type.
2. **Decoder:** Reconstructs the input data from the latent representation. It mirrors the encoder's architecture to produce output with the same dimensions as the input.

For text data, autoencoders typically use:

- Embedding Layers for converting words or tokens into dense vector representations.
- Recurrent Neural Networks (RNNs) such as LSTM or GRU units, due to their effectiveness in modeling sequential data.
- Dense Layers for dimensionality reduction and reconstruction tasks.



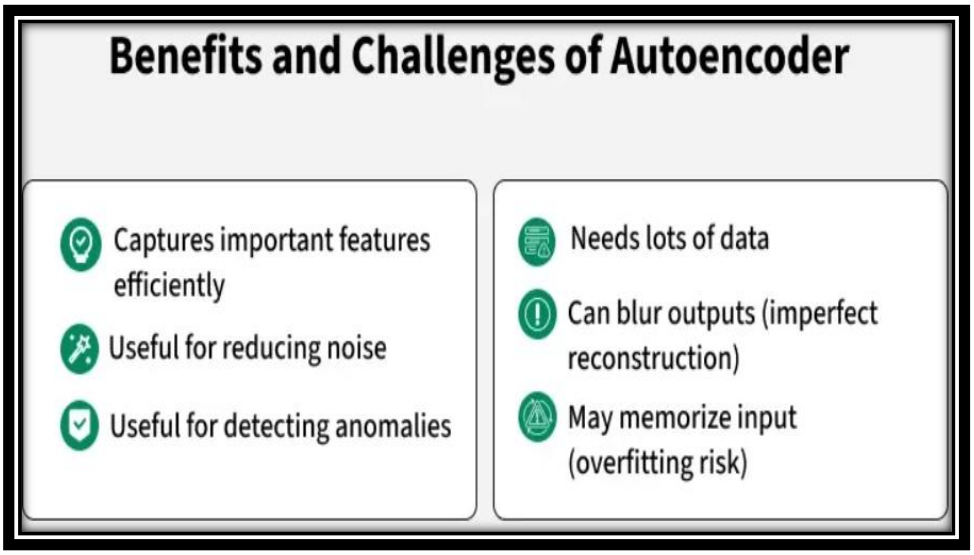
The model is trained using a reconstruction loss function (commonly Mean Squared Error or Cross-Entropy) that penalizes the difference between the original input and its reconstruction.



Use of Autoencoders in text datasets:

When applied to text data, autoencoders are valuable for:

- Learning compressed textual representations (latent embeddings)
- Preprocessing for tasks such as clustering or classification
- Noise reduction in textual inputs



Dataset used:

The **Pistachio 28-Features Dataset** consists of 28 numerical features capturing physical and morphological properties of pistachio nuts, such as length, width, area, perimeter, and color metrics. Labeled by type—**Kirmizi** and **Siit**—it provides a structured dataset suitable for machine learning tasks like dimensionality reduction, autoencoder-based feature extraction, anomaly detection, and classification. This dataset enables analysis of nut quality, type differentiation, and data-driven decision-making in agricultural research.

Source Code:

```
import os, zipfile
import pandas as pd
import numpy as np
from google.colab import drive
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.optimizers import Adam
import tensorflow as tf

SEED = 42
np.random.seed(SEED)
tf.random.set_seed(SEED)

drive.mount('/content/drive', force_remount=True)

zip_path = "/content/drive/MyDrive/FDL 5th Sem Practical Codes/PISTACHIO DATASET.zip"
extract_dir = "/content/pistachio_data"
os.makedirs(extract_dir, exist_ok=True)
with zipfile.ZipFile(zip_path, 'r') as z:
    z.extractall(extract_dir)

dataset_path = os.path.join(extract_dir, "Pistachio_Dataset", "Pistachio_28_Features_Dataset",
                            "Pistachio_28_Features_Dataset.xlsx")
df = pd.read_excel(dataset_path)

X = df.drop(columns=["Class"])
y = LabelEncoder().fit_transform(df["Class"])

X_scaled = StandardScaler().fit_transform(X)

X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=SEED, stratify=y)
```

```
input_dim = X_train.shape[1]
encoding_dim = 14
inp = Input(shape=(input_dim,))
enc = Dense(encoding_dim, activation='relu')(inp)
dec = Dense(input_dim, activation='sigmoid')(enc)
autoencoder = Model(inp, dec)
autoencoder.compile(optimizer=Adam(0.001), loss='mse')
autoencoder.fit(X_train, X_train, epochs=20, batch_size=32, validation_data=(X_test, X_test), verbose=1)

encoder = Model(inp, enc)
Z_train = encoder.predict(X_train)
Z_test = encoder.predict(X_test)

clf_raw = LogisticRegression(max_iter=200, random_state=SEED).fit(X_train, y_train)
clf_latent = LogisticRegression(max_iter=200, random_state=SEED).fit(Z_train, y_train)

acc_raw = accuracy_score(y_test, clf_raw.predict(X_test))
acc_latent = accuracy_score(y_test, clf_latent.predict(Z_test))

print(f"Test Reconstruction Loss: {autoencoder.evaluate(X_test, X_test):.4f}")
print(f"Accuracy (Raw features): {acc_raw:.4f}")
print(f"Accuracy (Latent features): {acc_latent:.4f}")
print("Sample latent vectors (first 3 rows):")
print(Z_test[:3])
```

Output:

```
Mounted at /content/drive
Epoch 1/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 1s 7ms/step - loss: 1.2114 - val_loss: 1.1640
Epoch 2/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 1.0836 - val_loss: 1.0293
Epoch 3/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.9554 - val_loss: 0.9196
Epoch 4/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.8590 - val_loss: 0.8555
Epoch 5/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.8000 - val_loss: 0.8149
Epoch 6/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.7610 - val_loss: 0.7861
Epoch 7/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.7329 - val_loss: 0.7642
Epoch 8/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.7114 - val_loss: 0.7478
Epoch 9/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6952 - val_loss: 0.7357
Epoch 10/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6834 - val_loss: 0.7265
```

```
Epoch 11/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6745 - val_loss: 0.7193
Epoch 12/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6675 - val_loss: 0.7135
Epoch 13/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6618 - val_loss: 0.7087
Epoch 14/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6571 - val_loss: 0.7047
Epoch 15/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6531 - val_loss: 0.7012
Epoch 16/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 0.6497 - val_loss: 0.6982
Epoch 17/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6467 - val_loss: 0.6955
Epoch 18/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6440 - val_loss: 0.6929
Epoch 19/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6416 - val_loss: 0.6907
Epoch 20/20
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6394 - val_loss: 0.6885
54/54 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step
14/14 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.6467
Test Reconstruction Loss: 0.6885
Accuracy (Raw features): 0.9093
Accuracy (Latent features): 0.8674
Sample latent vectors (first 3 rows):
[[12.22244     0.          7.532537    0.          1.2534617   0.
   0.         10.285851    0.          1.1022091   3.864142   11.848122
   0.          2.3860848 ]
 [ 0.9436313   0.          3.9727337   4.5880175   0.15188646  1.2236302
   0.          1.7601125   0.          0.          6.644242    6.8444605
   0.          0.        ]
 [ 0.          7.54191     0.          0.          0.          3.6766768
   0.          2.7069268   1.1893922   0.9167059   0.7033074   2.4570572
   3.5811048   0.        ]]
```

Learning Outcome:

EXPERIMENT 5

Aim: Application of Autoencoders on Image Dataset.

Objectives:

The objective of this experiment is to implement and evaluate an autoencoder on the MNIST handwritten digit dataset to learn compact latent representations of image data. The study aims to compress high-dimensional input images into a lower-dimensional encoding while retaining essential visual features, enabling effective reconstruction, dimensionality reduction, and potential use in downstream tasks such as classification or anomaly detection.

Theory:

Autoencoders are unsupervised neural networks that learn efficient representations of data by compressing inputs into a lower-dimensional latent space and reconstructing them. This experiment applies an autoencoder to the MNIST handwritten digit dataset, aiming to capture essential features of 28×28 grayscale images, reduce dimensionality, and accurately reconstruct the digits, demonstrating the network's capability for feature learning and potential use in classification or visualization tasks.

Architecture of Autoencoder:

An autoencoder's architecture consists of three main components that work together to compress and then reconstruct data which are as follows:

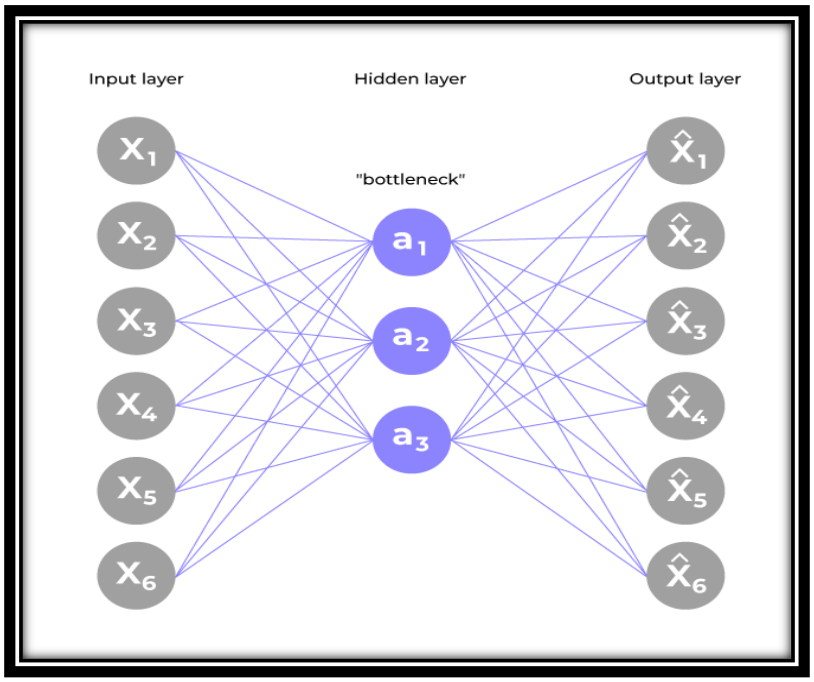
1. Encoder

It compresses the input data into a smaller, more manageable form by reducing its dimensionality while preserving important information. It has three layers which are:

- **Input Layer**: This is where the original data enters the network. It can be images, text features or any other structured data.
- **Hidden Layers**: These layers perform a series of transformations on the input data. Each hidden layer applies weights and activation functions to capture important patterns, progressively reducing the data's size and complexity.
- **Output (Latent Space)**: The encoder outputs a compressed vector known as the latent representation or encoding. This vector captures the important features of the input data in a condensed form helps in filtering out noise and redundancies.

2. Bottleneck (Latent Space)

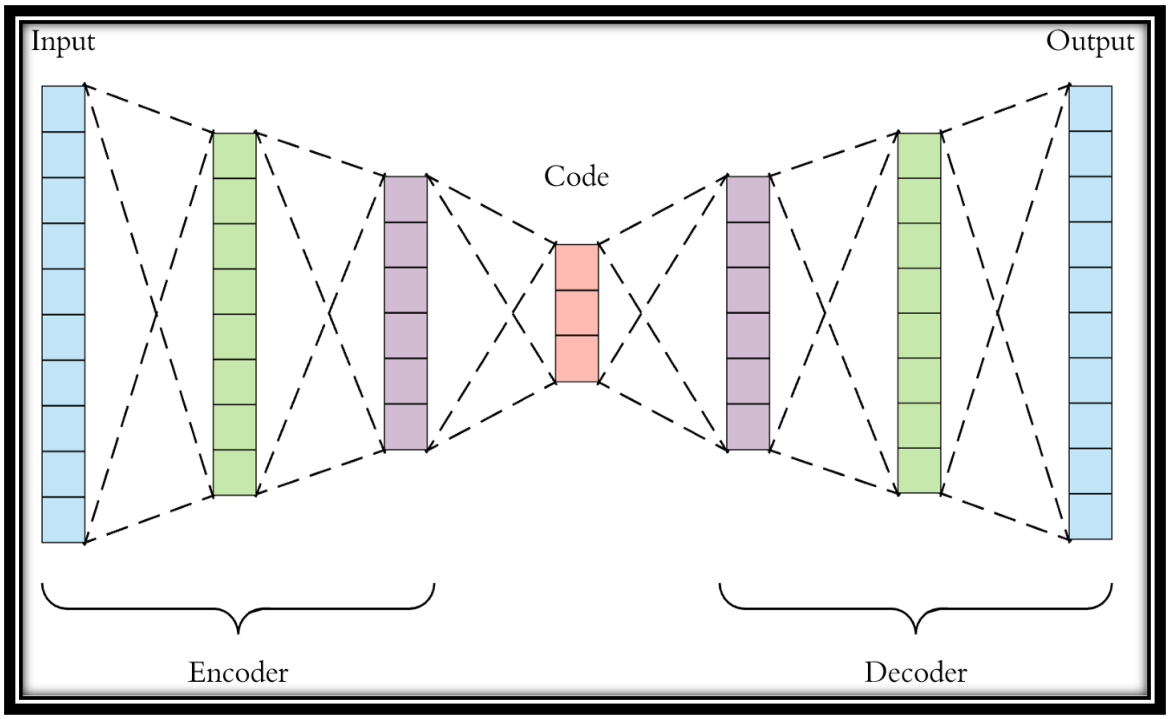
It is the smallest layer of the network which represents the most compressed version of the input data. It serves as the information bottleneck which force the network to prioritize the most significant features. This compact representation helps the model learn the underlying structure and key patterns of the input helps in enabling better generalization and efficient data encoding.



3. Decoder

It is responsible for taking the compressed representation from the latent space and reconstructing it back into the original data form.

- **Hidden Layers**: These layers progressively expand the latent vector back into a higher-dimensional space. Through successive transformations decoder attempts to restore the original data shape and details
- **Output Layer**: The final layer produces the reconstructed output which aims to closely resemble the original input. The quality of reconstruction depends on how well the encoder-decoder pair can minimize the difference between the input and output during training.



Autoencoders for Dimensionality Reduction:

Autoencoders reduce high-dimensional data into a lower-dimensional latent space, preserving essential features while discarding redundancies. This compact representation enables efficient storage, faster computation, and improved performance in downstream tasks such as clustering, visualization, or classification, making autoencoders a powerful tool for dimensionality reduction in complex datasets like images.

Advantages and Disadvantages of Autoencoders:

Autoencoders provide efficient feature extraction, dimensionality reduction, and unsupervised learning capabilities. They can compress data while retaining key information. However, they may require careful tuning, are prone to overfitting, and standard autoencoders cannot handle unseen data distributions well. They also do not inherently provide interpretability of latent features.

Implementation of Autoencoders:

In this experiment, a simple feedforward autoencoder was implemented on the MNIST dataset. Images were flattened to 784-dimensional vectors, encoded into a 64-dimensional latent space using a dense layer, and reconstructed with a decoder layer. The model was trained to minimize reconstruction error, allowing the network to learn meaningful features without supervision.

Dataset Used:

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0–9) with size 28×28 pixels. It is widely used for benchmarking image processing and machine learning algorithms. Each image represents a single digit, providing a simple yet effective dataset for testing autoencoder performance in reconstruction and feature learning tasks.

Source Code:

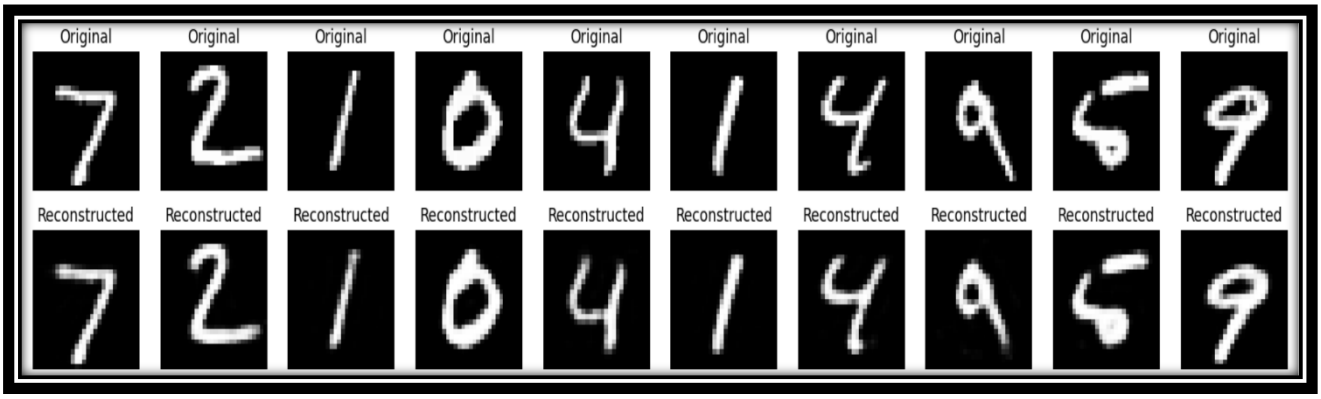
```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
import matplotlib.pyplot as plt

(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 = x_train.reshape((len(x_train), 28*28))
x_test = x_test.reshape((len(x_test), 28*28))
```

```
input_dim = 784
encoding_dim = 64

input_layer = Input(shape=(input_dim,))
encoded = Dense(encoding_dim, activation='relu')(input_layer)
decoded = Dense(input_dim, activation='sigmoid')(encoded)

autoencoder = Model(input_layer, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=20,
                batch_size=256,
                shuffle=True,
                validation_data=(x_test, x_test))

encoded_imgs = autoencoder.predict(x_test)
decoded_imgs = encoded_imgs.reshape((-1, 28, 28))

n = 10
plt.figure(figsize=(20, 4))
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], cmap="gray")
    plt.title("Reconstructed")
    plt.axis("off")

plt.show()
```

Output:

```
Epoch 1/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 12ms/step - loss: 0.3469 - val_loss: 0.1608
Epoch 2/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.1514 - val_loss: 0.1259
Epoch 3/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 16ms/step - loss: 0.1228 - val_loss: 0.1084
Epoch 4/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - loss: 0.1069 - val_loss: 0.0971
Epoch 5/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - loss: 0.0964 - val_loss: 0.0897
```

```
Epoch 6/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0894 - val_loss: 0.0847
Epoch 7/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0846 - val_loss: 0.0813
Epoch 8/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - loss: 0.0816 - val_loss: 0.0789
Epoch 9/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - loss: 0.0792 - val_loss: 0.0774
Epoch 10/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0777 - val_loss: 0.0762
Epoch 11/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 16ms/step - loss: 0.0768 - val_loss: 0.0754
Epoch 12/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0759 - val_loss: 0.0747
Epoch 13/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0755 - val_loss: 0.0743
Epoch 14/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0749 - val_loss: 0.0740
Epoch 15/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - loss: 0.0747 - val_loss: 0.0738
Epoch 16/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - loss: 0.0744 - val_loss: 0.0735
Epoch 17/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 0.0741 - val_loss: 0.0733
Epoch 18/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 12ms/step - loss: 0.0739 - val_loss: 0.0733
Epoch 19/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0738 - val_loss: 0.0731
Epoch 20/20
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - loss: 0.0738 - val_loss: 0.0730
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
```



Learning Outcome:

EXPERIMENT 6

Aim: Improving Autoencoders performance using Convolutional Layers in python (MNIST Dataset)

Objectives:

The objective of this experiment is to improve the performance of traditional autoencoders by incorporating convolutional layers, using MNIST dataset as a case study. While basic autoencoders use fully connected layers that may not effectively capture spatial feature in image data, convolutional autoencoders (CAEs) utilize convolution and pooling operations to preserve spatial structure and learn more meaningful representations. This experiment focuses on building and training a convolutional autoencoder that can efficiently encode and reconstruct hand-written data images from the MNIST Dataset. The goal is to observe how convolutional layers enhance the autoencoder's ability to extract important features, reduce reconstruction error, and achieve better performance in tasks such as denoising and image compression. Through this, we will understand the advantages of using convolutional architectures for image-based deep learning applications.

Theory:

1. Introduction:

This experiment aims to enhance autoencoder performance by integrating convolutional layers, leveraging spatial feature extraction for improved image reconstruction on the MNIST dataset.

2. Autoencoders and their purpose:

Autoencoders are neural networks designed to learn efficient data representations by encoding inputs into a compressed latent space and decoding them back, primarily for tasks like dimensionality reduction and noise removal.

3. Feature-driven learning:

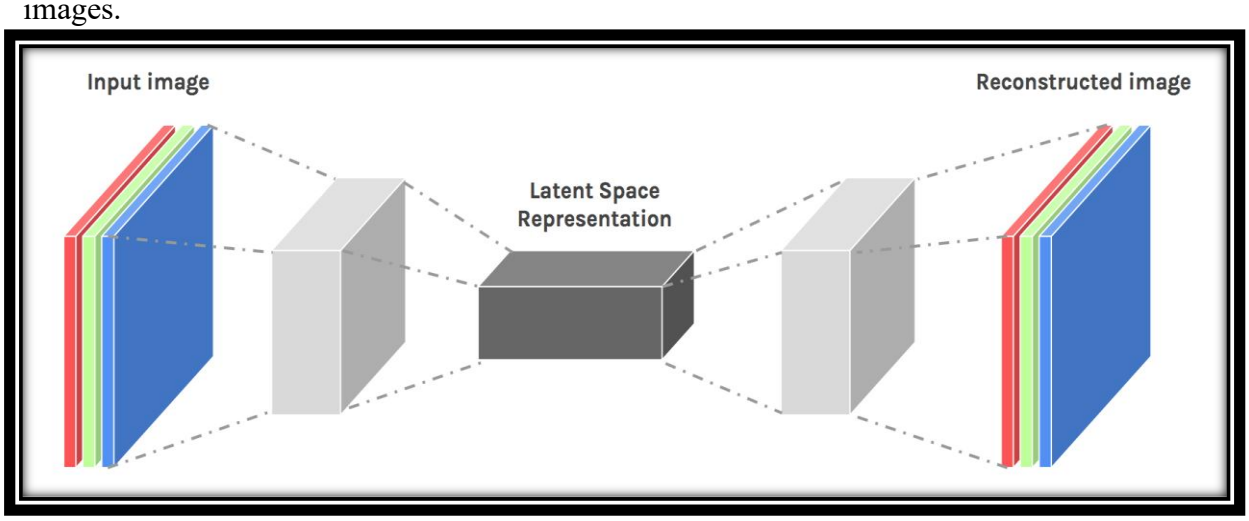
Convolutional layers capture local spatial features such as edges and patterns, enabling the autoencoder to learn richer, more meaningful representations compared to fully connected layers.

4. MNIST Dataset:

The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0–9) with size 28×28 pixels. It is widely used for benchmarking image processing and machine learning algorithms. Each image represents a single digit, providing a simple yet effective dataset for testing autoencoder performance in reconstruction and feature learning tasks.

5. Model Architecture:

The convolutional autoencoder employs convolutional and pooling layers in the encoder to extract features, with symmetrical deconvolutional layers in the decoder to reconstruct the input images.



6. Training and Loss function:

The model is trained using the Mean Squared Error (MSE) loss function, optimizing reconstruction accuracy through backpropagation with an optimizer like Adam.

7. Results and Visualization:

Training results demonstrate improved reconstruction quality, visualized by comparing original and decoded images, showcasing clearer digit structures and reduced noise.

8. Future Improvements and Conclusion:

Future work may include deeper architectures, regularization techniques, or variational autoencoders to further enhance feature learning and robustness, confirming convolutional layers' effectiveness in autoencoders.

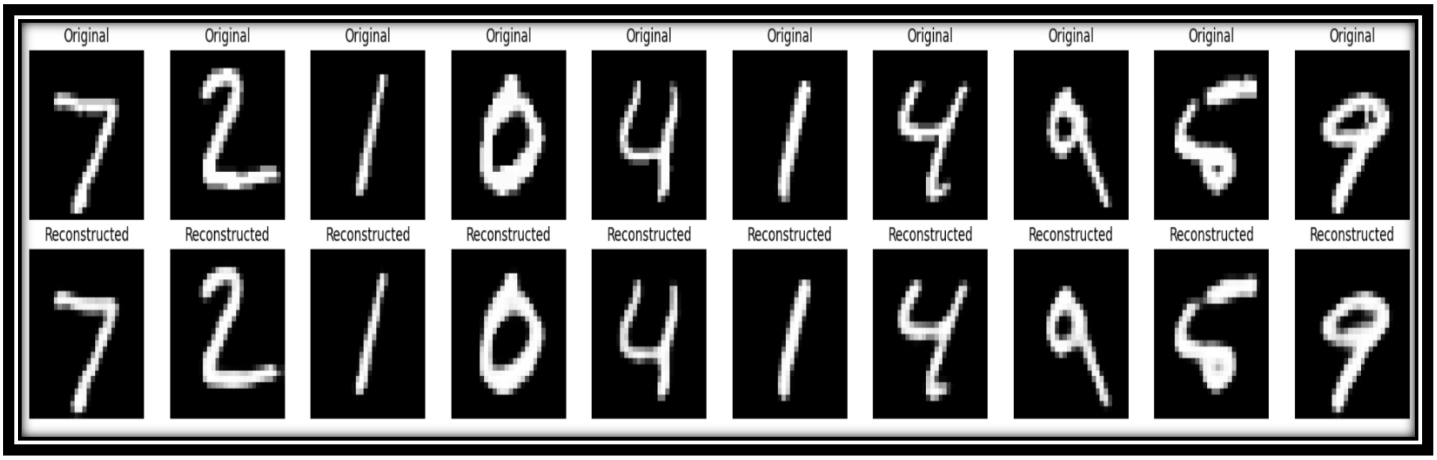
Source Code:

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


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

encoder_input = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu", padding="same")(encoder_input)
x = layers.MaxPooling2D(2, padding="same")(x)
x = layers.Conv2D(16, 3, activation="relu", padding="same")(x)
encoded = layers.MaxPooling2D(2, padding="same")(x)  # [B, 7, 7, 16]
```

```
x = layers.Conv2D(16, 3, activation="relu", padding="same")(encoded)
x = layers.UpSampling2D(2)(x)
x = layers.Conv2D(32, 3, activation="relu", padding="same")(x)
x = layers.UpSampling2D(2)(x)
decoded = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)  # back to [28, 28, 1]

autoencoder = keras.Model(encoder_input, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

autoencoder.fit(x_train, x_train,
                epochs=10,
                batch_size=128,
                shuffle=True,
                validation_split=0.1)
encoder = keras.Model(encoder_input, encoded)
encoded_imgs = encoder.predict(x_test)
decoded_imgs = autoencoder.predict(x_test)
n = 10
plt.figure(figsize=(20, 4))
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.tight_layout()
plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Epoch 1/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 86s 198ms/step - loss: 0.2619 - val_loss: 0.0832
Epoch 2/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 84s 199ms/step - loss: 0.0800 - val_loss: 0.0769
Epoch 3/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 141s 197ms/step - loss: 0.0754 - val_loss: 0.0744
Epoch 4/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 144s 202ms/step - loss: 0.0734 - val_loss: 0.0729
Epoch 5/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 85s 201ms/step - loss: 0.0720 - val_loss: 0.0719
Epoch 6/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 143s 203ms/step - loss: 0.0712 - val_loss: 0.0714
Epoch 7/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 83s 198ms/step - loss: 0.0706 - val_loss: 0.0708
Epoch 8/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 143s 201ms/step - loss: 0.0699 - val_loss: 0.0703
Epoch 9/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 141s 199ms/step - loss: 0.0696 - val_loss: 0.0699
Epoch 10/10
422/422 ━━━━━━━━━━━━━━━━━━━━ 143s 202ms/step - loss: 0.0693 - val_loss: 0.0697
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step
313/313 ━━━━━━━━━━━━━━━━━━━━ 5s 15ms/step
```



Learning Outcome:

EXPERIMENT 7

Aim: Implementation of RNN model for Stock Price Prediction in Python.

Objective:

The objective of this experiment is to design and implement a Recurrent Neural Network (RNN) model in Python for predicting stock prices using historical stock market data. Stock prices are inherently sequential and time-dependant, making RNN's a suitable choice due to their ability to retain information from previous time steps through internal memory. This experiment aims to explore how RNN's can model complex temporal relationships in stock price data and generate accurate predictions of future values. The implementation will involve data preprocessing, feature scaling, sequence generation, and training the RNN on real stock datasets. The experiment also seeks to evaluate the model's performance using appropriate error metrics and visualize prediction trends against actual stock prices. Ultimately, this experiment provides hands-on experience with time series forecasting using deep learning and highlights the practical use of RNN's in financial market analysis and decision making.

Theory:

Introduction to RNN:

Recurrent Neural Networks (RNNs) are a class of artificial neural networks designed for processing sequential data, such as time series, text, or speech. Unlike traditional feedforward networks, RNNs have loops that allow information to persist across time steps. This makes them ideal for problems where past context influences future predictions. In time series forecasting, like stock price prediction, RNNs are particularly useful because they can learn trends, seasonality, and temporal dependencies in the data. However, vanilla RNNs struggle with long-term memory due to vanishing gradients, which is why advanced versions like LSTM (Long Short-Term Memory) are often used alongside.

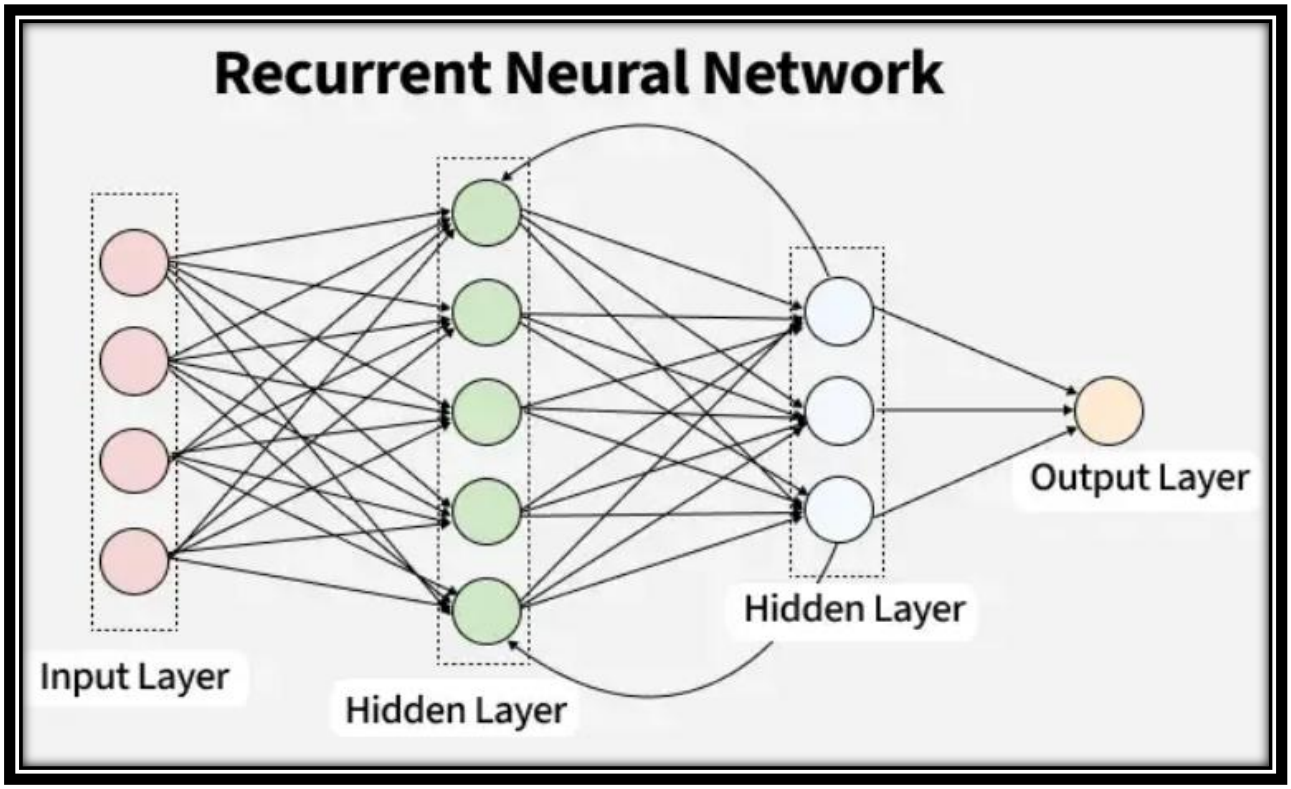
How RNN works:

RNNs process sequences one step at a time, maintaining a hidden state that captures information from previous time steps. At each time step, the input and the previous hidden state are combined and passed through an activation function to generate a new hidden state. This recursive process allows RNNs to "remember" information over time. The output can be generated at each step or after the entire sequence. However, during training, gradients can vanish or explode when backpropagated through many time steps, making it hard for vanilla RNNs to learn long-term dependencies. Variants like LSTM and GRU were developed to address this.

Layers of RNN:

RNNs can be stacked to form deep recurrent networks, improving their capacity to learn complex patterns. A basic RNN layer consists of units with recurrent connections. Each RNN unit takes an input vector and the previous hidden state to produce a new hidden state. In deeper architectures, multiple RNN layers are stacked, where each layer processes the entire

sequence and passes its outputs to the next. This project uses two SimpleRNN layers to capture short-term dependencies and one LSTM layer for long-term learning. Dropout layers are added between RNN layers to prevent overfitting by randomly deactivating neurons during training.



LSTM's:

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) designed to overcome the vanishing gradient problem of traditional RNNs, enabling them to capture long-term dependencies in sequential data. LSTMs achieve this by using special structures called gates (input, forget, and output gates) that regulate the flow of information, deciding what to keep, update, or discard from the cell state. This gating mechanism allows LSTMs to remember important information over many time steps, making them highly effective for time series forecasting, natural language processing, and other sequence modeling tasks such as stock price prediction.

Dataset used:

The dataset used for this experiment consists of daily closing prices of Apple Inc. (AAPL) stock from January 2015 to December 2023. This data was obtained using the yfinance Python library, which fetches historical stock market data directly from Yahoo Finance. We focus only on the "Close" column, which reflects the final price of the stock for each trading day. This time series is ideal for forecasting problems since it contains long-term trends, seasonality, and market noise. Any missing values are dropped to maintain data integrity. This dataset serves as a real-world example for testing the performance of RNN models.

Code Implementation:

The implementation involves several steps: First, the stock data is loaded and scaled between 0 and 1 using MinMaxScaler to normalize input values. Next, sequences of 60 consecutive closing prices are created, where the 61st value is the target output. This transforms the data into a supervised learning problem. The dataset is then split into 80% training and 20% testing data. The model consists of two SimpleRNN layers followed by one LSTM layer and a Dense output. It is compiled with the Adam optimizer and MSE loss. Evaluation is done using MSE, MAE, and a percentage-based accuracy estimate.

Source Code:

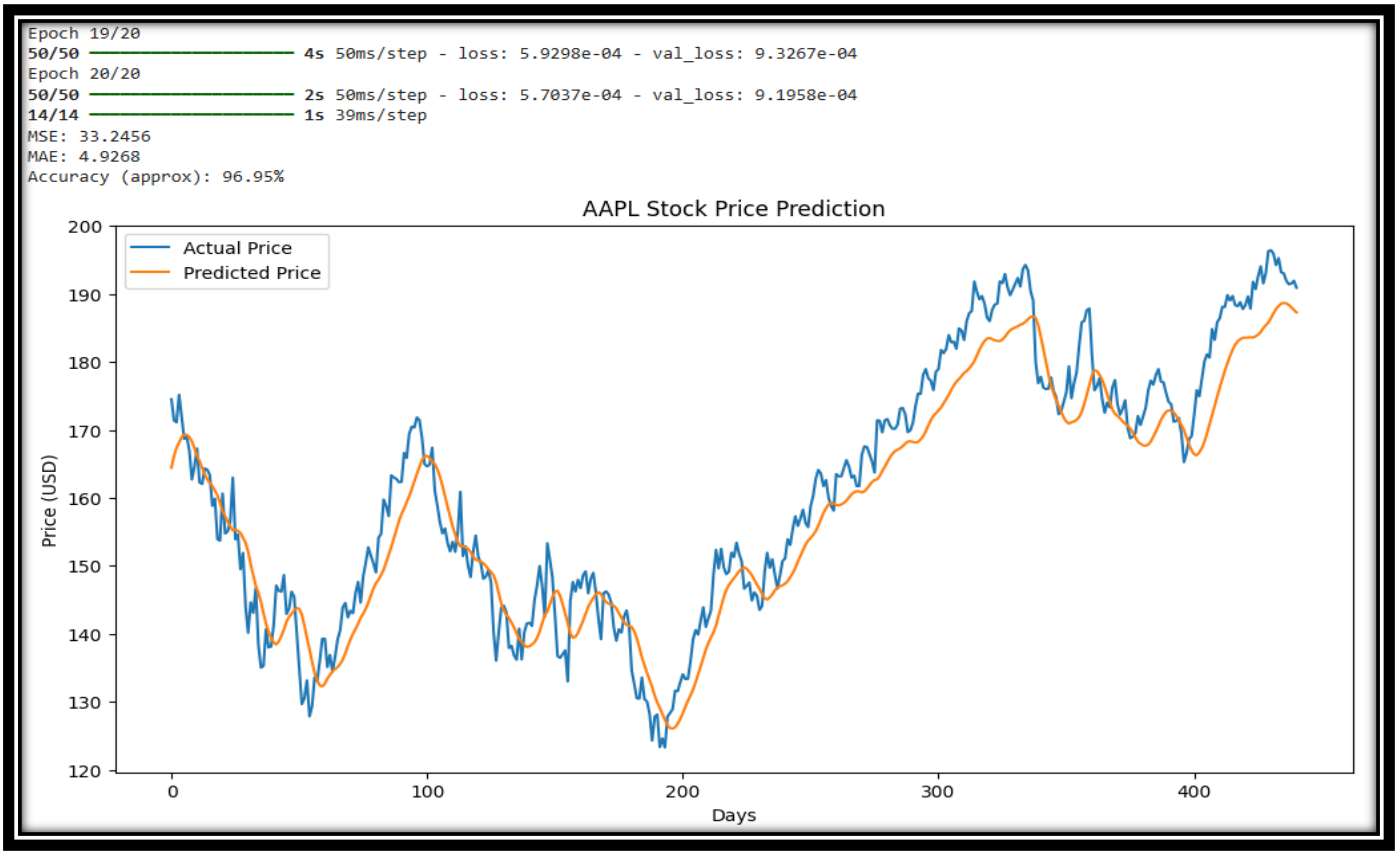
```
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM,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

data = yf.download('AAPL', start='2015-01-01', end='2023-12-31')
close_prices = data[['Close']].dropna()
scaler = MinMaxScaler()
scaled_prices = scaler.fit_transform(close_prices)

def create_sequences(data, seq_len):
    X, y = [], []
    for i in range(seq_len, len(data)):
        X.append(data[i - seq_len:i, 0])
        y.append(data[i, 0])
    return np.array(X), np.array(y)

seq_len = 60
X, y = create_sequences(scaled_prices, seq_len)
X = X.reshape((X.shape[0], X.shape[1], 1))
split_index = int(len(X) * 0.8)
X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]
```

```
model = Sequential([
    LSTM(50, activation='tanh', return_sequences=True, input_shape=(seq_len, 1)),
    Dropout(0.2),
    LSTM(50, activation='tanh', return_sequences=False),
    Dropout(0.2),
    Dense(1)
])
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.1, verbose=1)

predicted = model.predict(X_test)
predicted_inv = scaler.inverse_transform(predicted)
actual_inv = scaler.inverse_transform(y_test.reshape(-1, 1))

mse = mean_squared_error(actual_inv, predicted_inv)
mae = mean_absolute_error(actual_inv, predicted_inv)
acc = 100 - (np.mean(np.abs((actual_inv - predicted_inv) / actual_inv)) * 100)

print(f"MSE: {mse:.4f}")
print(f"MAE: {mae:.4f}")
print(f"Accuracy (approx): {acc:.2f}%")

plt.figure(figsize=(12,6))
plt.plot(actual_inv, label='Actual Price')
plt.plot(predicted_inv, label='Predicted Price')
plt.title('AAPL Stock Price Prediction')
plt.xlabel('Days')
plt.ylabel('Price (USD)')
plt.legend()
plt.show()
```

Output:

```
/tmp/ipython-input-1009289260.py:10: FutureWarning: YF.download() has changed argument auto_adjust default to True
  data = yf.download('AAPL', start='2015-01-01', end='2023-12-31')
[*********************100%**********************]  1 of 1 completedEpoch 1/20

/usr/local/lib/python3.12/dist-packages/keras/src/layers/rnn/rnn.py:199: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the firs
  super().__init__(**kwargs)
50/50 ━━━━━━━━━━━━━━━━━━━━ 7s 60ms/step - loss: 0.0157 - val_loss: 0.0023
Epoch 2/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 50ms/step - loss: 9.3160e-04 - val_loss: 0.0013
Epoch 3/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 50ms/step - loss: 0.0010 - val_loss: 0.0018
Epoch 4/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 57ms/step - loss: 8.2108e-04 - val_loss: 9.9509e-04
Epoch 5/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 50ms/step - loss: 7.6280e-04 - val_loss: 0.0014
Epoch 6/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 50ms/step - loss: 7.2039e-04 - val_loss: 0.0021
Epoch 7/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 2s 49ms/step - loss: 7.2337e-04 - val_loss: 8.8628e-04
Epoch 8/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 64ms/step - loss: 7.4817e-04 - val_loss: 0.0012
Epoch 9/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 57ms/step - loss: 7.1420e-04 - val_loss: 0.0011
Epoch 10/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 2s 49ms/step - loss: 6.6785e-04 - val_loss: 0.0011
Epoch 11/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 53ms/step - loss: 6.8108e-04 - val_loss: 8.8763e-04
Epoch 12/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 52ms/step - loss: 6.6059e-04 - val_loss: 0.0044
Epoch 13/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 51ms/step - loss: 7.5310e-04 - val_loss: 0.0025
Epoch 14/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 6s 69ms/step - loss: 9.5437e-04 - val_loss: 0.0013
Epoch 15/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 60ms/step - loss: 6.6488e-04 - val_loss: 0.0036
Epoch 16/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 50ms/step - loss: 8.6255e-04 - val_loss: 0.0012
Epoch 17/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 2s 49ms/step - loss: 5.7882e-04 - val_loss: 8.3025e-04
Epoch 18/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 71ms/step - loss: 5.6181e-04 - val_loss: 0.0017
```

```
Epoch 19/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 50ms/step - loss: 5.9298e-04 - val_loss: 9.3267e-04
Epoch 20/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 2s 50ms/step - loss: 5.7037e-04 - val_loss: 9.1958e-04
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 39ms/step
MSE: 33.2456
MAE: 4.9268
Accuracy (approx): 96.95%
```



Learning Outcome:

EXPERIMENT 8

Aim: Using LSTM for prediction of future weather of cities in Python.

Objective:

The objective of this experiment is to predict daily average temperatures for Delhi, Mumbai, and Chennai using LSTM neural networks trained on historical weather data. The study evaluates model performance through loss and accuracy metrics and visualizes predictions to assess forecasting effectiveness.

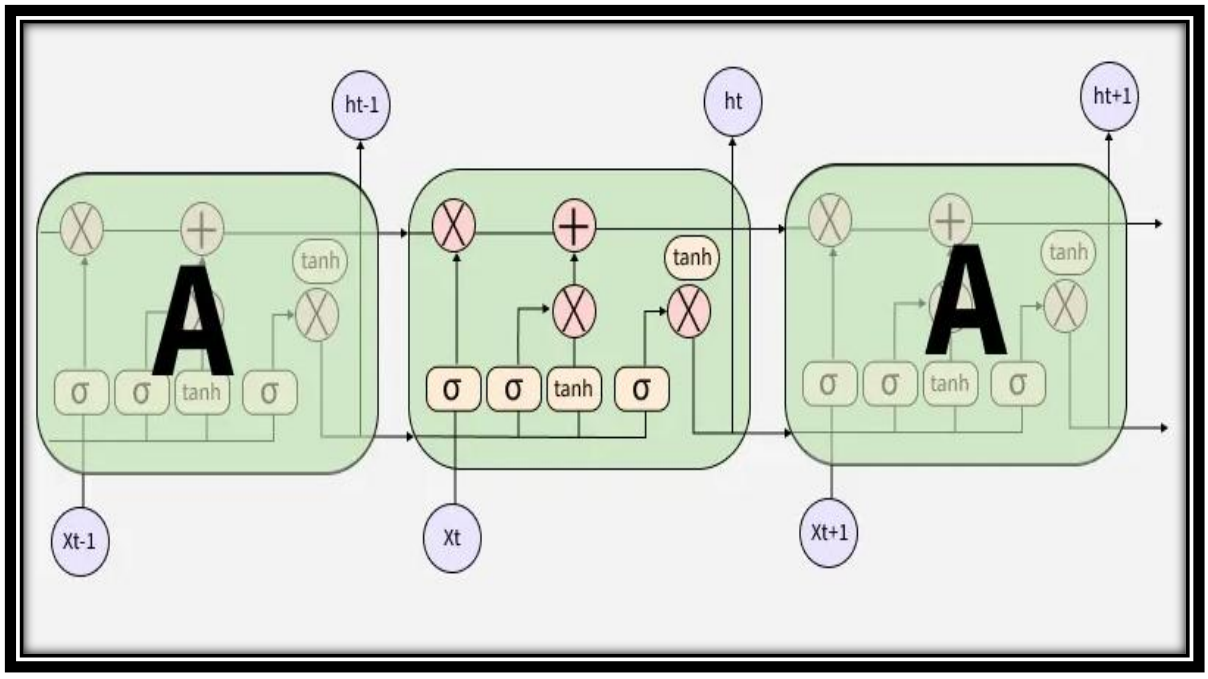
Theory:

RNN's:

Recurrent Neural Networks (RNNs) are a class of neural networks designed to handle sequential data by maintaining a memory of previous inputs through internal states. This makes them suitable for time series prediction tasks like weather forecasting. However, traditional RNNs suffer from the vanishing gradient problem, which limits their ability to learn long-term dependencies in sequences.

LSTM's and its working:

Long Short-Term Memory (LSTM) networks are a special type of RNN specifically designed to overcome the limitations of standard RNNs. LSTMs use memory cells and gating mechanisms—input, forget, and output gates—to selectively retain or discard information over long sequences. This structure allows LSTMs to capture long-term dependencies more effectively, making them powerful for modeling complex temporal patterns such as weather data.



Why use LSTM:

Weather data is inherently sequential and influenced by patterns over various time scales. LSTM networks excel at learning such temporal dependencies, enabling more accurate forecasts compared to traditional models. Their ability to remember relevant past information and ignore irrelevant details improves prediction quality, making LSTM an ideal choice for temperature forecasting.

Dataset Used:

The dataset consists of historical daily weather records for three major Indian cities: Delhi, Mumbai, and Chennai. Each CSV file contains data spanning from 1990 to 2022, including average temperature (tavg), minimum temperature (tmin), maximum temperature (tmax), and precipitation (prcp). For this experiment, the focus is on the daily average temperature (tavg) as the target variable for prediction. Missing temperature values were handled using linear interpolation to maintain data continuity. This comprehensive dataset provides a robust basis for training and evaluating LSTM models for temperature forecasting.

Application in this experiment:

In this experiment, LSTM networks are trained on historical temperature data from three major Indian cities—Delhi, Mumbai, and Chennai. The model uses a sliding window approach with a 7-day lookback to predict the next day's average temperature. Performance is evaluated through loss metrics and accuracy, and predictions are visualized to compare actual versus forecasted values.

Source Code:

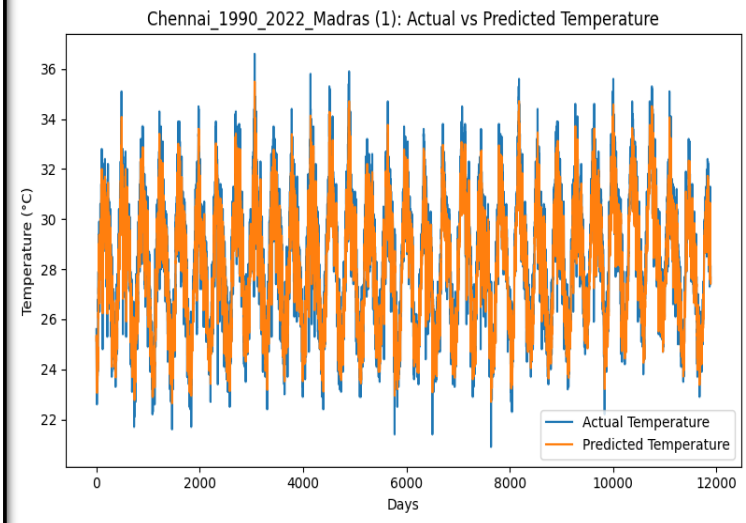
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from sklearn.preprocessing import MinMaxScaler
from google.colab import files
from sklearn.metrics import mean_absolute_error

print("Please upload CSV files for Delhi, Mumbai, and Chennai.")
uploaded = files.upload()

def load_and_preprocess_custom(filepath):
    df = pd.read_csv(filepath, parse_dates=['time'], dayfirst=True)
    df = df.sort_values('time')
    df['tavg'] = df['tavg'].interpolate(method='linear')
    df = df.dropna(subset=['tavg'])
    temperature = df['tavg'].values.reshape(-1, 1)
    scaler = MinMaxScaler()
    temp_scaled = scaler.fit_transform(temperature)
    X, y = [], []
    for i in range(7, len(temp_scaled)):
        X.append(temp_scaled[i-7:i])
        y.append(temp_scaled[i])
    X, y = np.array(X), np.array(y)
    return X, y, scaler

def build_and_train_lstm(X, y, epochs=20, batch_size=16):
    model = Sequential()
    model.add(LSTM(50, activation='relu', input_shape=(X.shape[1], X.shape[2])))
    model.add(Dense(1))
    model.compile(optimizer='adam', loss='mse')
    history = model.fit(X, y, epochs=epochs, batch_size=batch_size, verbose=1)
    return model, history
```

```python
def plot_predictions(actual, predicted, city_name):
    plt.figure(figsize=(8, 5))
    plt.plot(actual, label='Actual Temperature')
    plt.plot(predicted, label='Predicted Temperature')
    plt.xlabel('Days')
    plt.ylabel('Temperature (°C)')
    plt.title(f'{city_name}: Actual vs Predicted Temperature')
    plt.legend()
    plt.tight_layout()
    plt.show()

for filename in uploaded.keys():
    print(f"\nProcessing city data from file: {filename}")
    X, y, scaler = load_and_preprocess_custom(filename)
    model, history = build_and_train_lstm(X, y, epochs=20, batch_size=16)
    predicted_temp = model.predict(X)
    predicted_temp_rescaled = scaler.inverse_transform(predicted_temp)
    actual_temp_rescaled = scaler.inverse_transform(y.reshape(-1, 1))
    mae = mean_absolute_error(actual_temp_rescaled, predicted_temp_rescaled)
    mean_actual = np.mean(actual_temp_rescaled)
    accuracy = 100 * (1 - (mae / mean_actual))
    print(f"Final training loss (MSE): {history.history['loss'][-1]:.4f}")
    print(f"Accuracy: {accuracy:.2f}%")
    plot_predictions(actual_temp_rescaled, predicted_temp_rescaled, filename.split('.')[0])
```

Output:

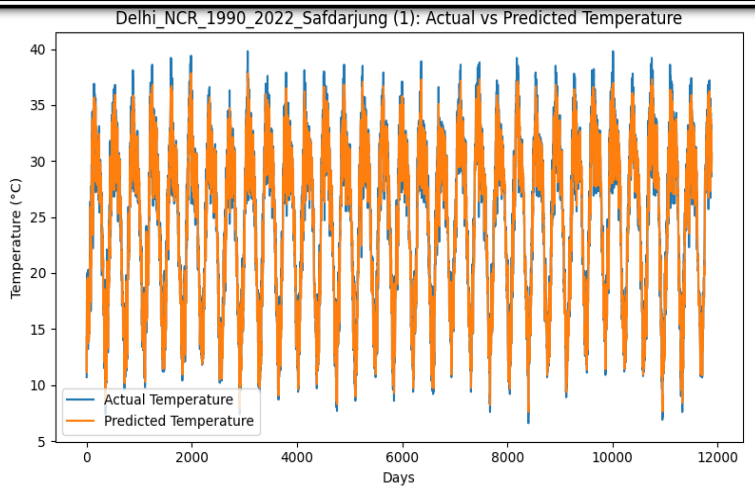
```
Please upload CSV files for Delhi, Mumbai, and Chennai.
Choose Files 3 files
Chennai_1990_2022_Madras.csv(text/csv) - 319933 bytes, last modified: 9/22/2025 - 100% done
Delhi_NCR_1990_2022_Safdarjung.csv(text/csv) - 322302 bytes, last modified: 9/22/2025 - 100% done
Mumbai_1990_2022_Santacruz.csv(text/csv) - 319190 bytes, last modified: 9/22/2025 - 100% done
Saving Chennai_1990_2022_Madras.csv to Chennai_1990_2022_Madras (1).csv
Saving Delhi_NCR_1990_2022_Safdarjung.csv to Delhi_NCR_1990_2022_Safdarjung (1).csv
Saving Mumbai_1990_2022_Santacruz.csv to Mumbai_1990_2022_Santacruz (1).csv

Processing city data from file: Chennai_1990_2022_Madras (1).csv
Epoch 1/20
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0204
Epoch 2/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0042
Epoch 3/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0036
Epoch 4/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0035
Epoch 5/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0033
Epoch 6/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0034
Epoch 7/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0034
Epoch 8/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0035
Epoch 9/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0033
Epoch 10/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0034
Epoch 11/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0034
Epoch 12/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0033
Epoch 13/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0032
Epoch 14/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0033
Epoch 15/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0034
Epoch 16/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0033
Epoch 17/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0034
```

```
Epoch 18/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0034
Epoch 19/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0035
Epoch 20/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0033
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Final training loss (MSE): 0.0033
Accuracy: 97.47%
```



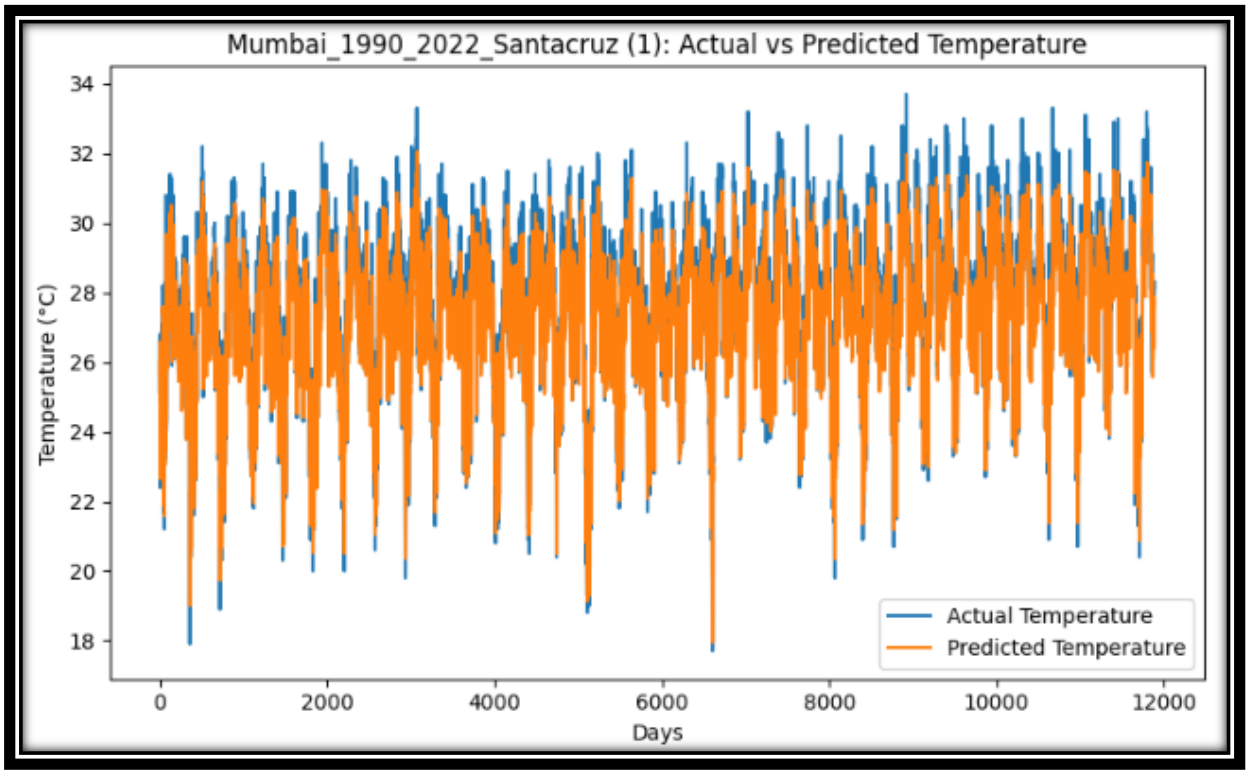

```
Processing city data from file: Delhi_NCR_1990_2022_Safdarjung (1).csv
Epoch 1/20
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - loss: 0.0403
Epoch 2/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0030
Epoch 3/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0026
```

```
Epoch 4/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0022
Epoch 5/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0022
Epoch 6/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0021
Epoch 7/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0021
Epoch 8/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.0021
Epoch 9/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0021
Epoch 10/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0022
Epoch 11/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - loss: 0.0022
Epoch 12/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0021
Epoch 13/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0021
Epoch 14/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0021
Epoch 15/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0021
Epoch 16/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0022
Epoch 17/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0021
Epoch 18/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0021
Epoch 19/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0021
Epoch 20/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0021
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step
Final training loss (MSE): 0.0021
Accuracy: 95.45%
```




```
Processing city data from file: Mumbai_1990_2022_Santacruz (1).csv
Epoch 1/20
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0498
Epoch 2/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0040
Epoch 3/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - loss: 0.0033
Epoch 4/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0031
Epoch 5/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0032
Epoch 6/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0031
Epoch 7/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0031
```

```
Epoch 8/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0032
Epoch 9/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0032
Epoch 10/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - loss: 0.0031
Epoch 11/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0031
Epoch 12/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0031
Epoch 13/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - loss: 0.0031
Epoch 14/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0031
Epoch 15/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0032
Epoch 16/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0030
Epoch 17/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - loss: 0.0031
Epoch 18/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s 5ms/step - loss: 0.0030
Epoch 19/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0032
Epoch 20/20
743/743 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - loss: 0.0032
372/372 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step
Final training loss (MSE): 0.0031
Accuracy: 97.24%
```



Learning Outcome:

EXPERIMENT 9

Aim: Implementation of Transfer Learning using the pre-trained model (MobileNetV2) for image classification in Python

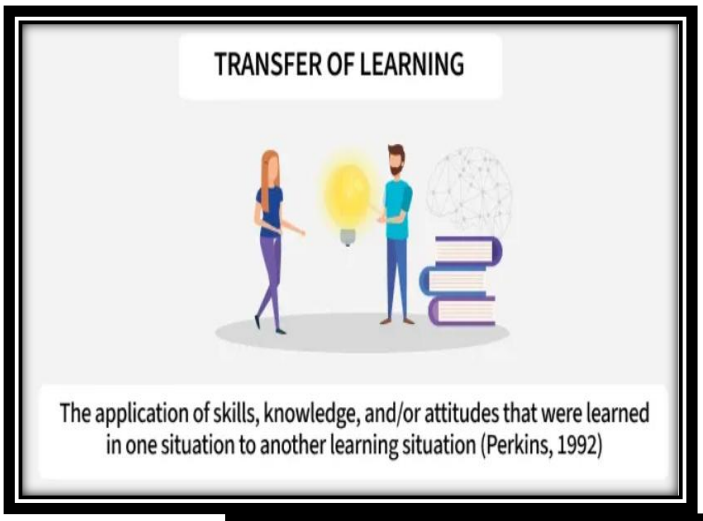
Objective:

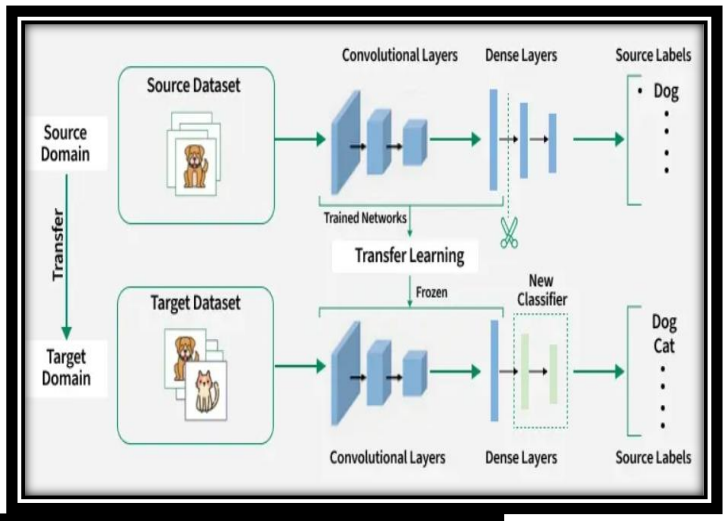
The objective of this experiment is to implement transfer learning using the pre-trained MobileNetV2 model for image classification tasks in Python. By leveraging the learned features of MobileNetV2, which has been trained on a large dataset, the goal is to efficiently classify new images with high accuracy while reducing the computational cost and training time compared to building a model from scratch.

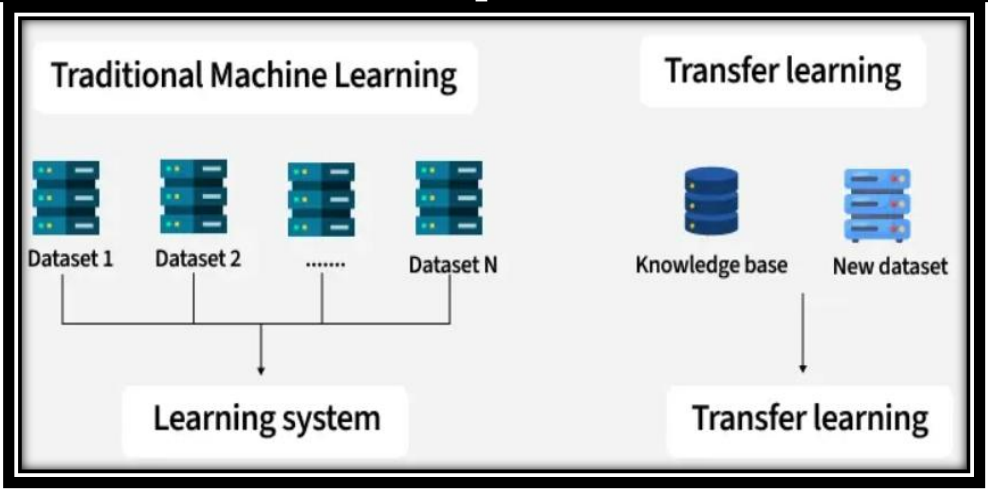
Theory:

Transfer Learning:

Transfer learning is a machine learning technique where a model developed for one task is reused as the starting point for a model on a different but related task. Instead of training a model from scratch, transfer learning leverages the knowledge (learned features and weights) from a pre-trained model—typically trained on a large dataset like ImageNet—and applies it to a new problem with comparatively less data. This approach significantly reduces training time, improves performance, and is especially useful when labelled data is scarce.





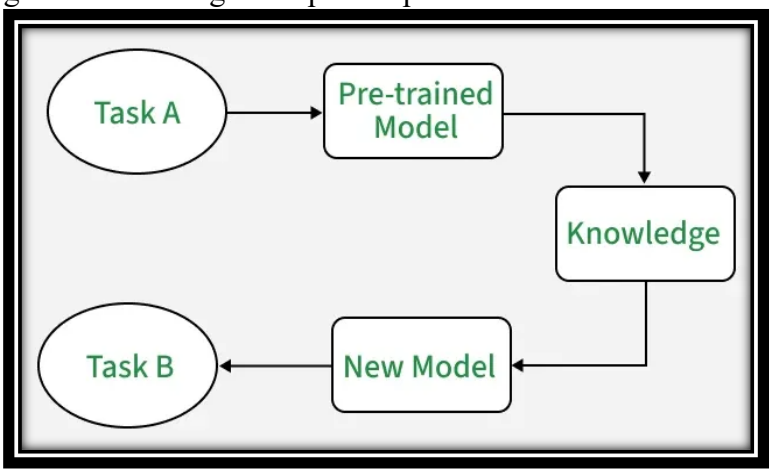


Importance of Transfer Learning:

Transfer learning is important because it enables faster and more efficient model development, particularly when working with limited data or computational resources. By reusing the learned features from a model trained on a large dataset, such as ImageNet, it reduces the need to train deep networks from scratch. This not only speeds up the training process but also improves accuracy and generalization on new tasks. It is especially valuable in domains like medical imaging, natural language processing, and facial recognition, where gathering large labelled datasets can be costly or impractical.

Working of Transfer Learning:

Transfer learning works by taking a pre-trained model—usually trained on a large-scale dataset—and adapting it to a new, related task. The initial layers of the model, which capture general features like edges and textures, are typically frozen to retain their learned weights. Then, new layers specific to the target task are added on top of the base model. These layers are trained on the new dataset, and optionally, some deeper layers of the base model may also be fine-tuned to further adapt to the new data. This process allows the model to leverage existing knowledge while learning task-specific patterns.



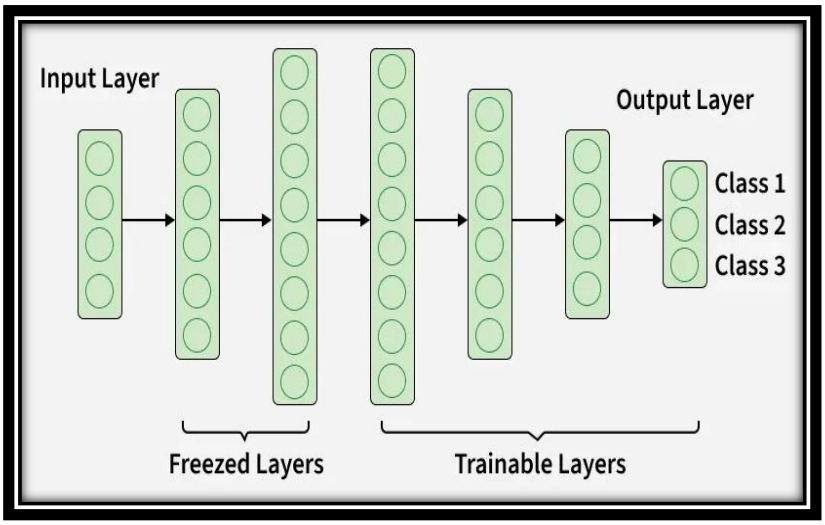
Frozen and Trainable Layers:

In transfer learning, frozen layers refer to those parts of the pre-trained model whose weights are kept constant during training on the new task. These layers typically capture generic, low-level features (like edges, textures, and shapes) that are useful across many visual tasks. Unfrozen or trainable layers, on the other hand, are allowed to update their weights during training. These are usually the newly added layers or sometimes deeper layers of the pre-trained model, which adapt to the specific characteristics of the new dataset. Freezing and unfreezing layers helps balance computational efficiency and model adaptability.

Deciding which layers to freeze or train/unfreeze:

The decision on which layers to freeze or train depends largely on the similarity between the original dataset (used to train the base model) and the new dataset. If the new task is closely related to the original task (e.g., both are natural image classification), freezing most of the base layers and training only the top layers may be sufficient. However, if the new task is significantly different, more layers—especially the deeper ones—should be unfrozen and fine-tuned to learn more task-specific features. Additionally, the size of the new dataset plays

a role; with limited data, freezing more layers helps prevent overfitting, whereas larger datasets allow for more layers to be trained effectively.



Dataset Used:

In this experiment, the MNIST dataset is used as the target dataset for transfer learning. MNIST is a widely-used benchmark dataset consisting of 70,000 grayscale images of handwritten digits ranging from 0 to 9, with 60,000 images for training and 10,000 for testing. Each image is originally of size 28×28 pixels, but since the MobileNetV2 model expects three-channel (RGB) images of a minimum size (typically 32×32 or higher), the images are pre-processed by stacking grayscale channels to create RGB images and resizing them to 32×32. Despite being simple and low-resolution, MNIST provides a useful case for demonstrating how transfer learning can be applied even to small, non-complex datasets by leveraging the representational power of pre-trained models.

Source Code and Output:

```
from tensorflow.keras.datasets import mnist
import numpy as np
import tensorflow as tf
from tensorflow.keras.utils import to_categorical


(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = np.stack([train_images]*3, axis=-1) / 255.0
test_images = np.stack([test_images]*3, axis=-1) / 255.0

train_images = tf.image.resize(train_images, [32, 32])
test_images = tf.image.resize(test_images, [32, 32])

train_labels = to_categorical(train_labels, 10)
test_labels = to_categorical(test_labels, 10)
```

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

base_model = MobileNetV2(weights='imagenet', include_top=False, input_shape=(32, 32, 3))
base_model.trainable = False  # Freeze base model

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)
```

```
/tmp/ipython-input-3104373990.py:5: UserWarning: `input_shape` is undefined or non-square, or `rows` is not in [96, 128, 160, 192, 224]. Weights for input shape (224, 224) will be loaded as the default.
  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/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224_no_top.h5
9406464/9406464 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=10, validation_split=0.2)
```

```
Epoch 1/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 28s 9ms/step - accuracy: 0.4287 - loss: 1.8500 - val_accuracy: 0.5996 - val_loss: 1.3018
Epoch 2/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.6114 - loss: 1.2596 - val_accuracy: 0.6392 - val_loss: 1.1415
Epoch 3/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.6341 - loss: 1.1389 - val_accuracy: 0.6526 - val_loss: 1.0739
Epoch 4/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.6470 - loss: 1.0803 - val_accuracy: 0.6598 - val_loss: 1.0375
Epoch 5/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.6525 - loss: 1.0431 - val_accuracy: 0.6661 - val_loss: 1.0147
Epoch 6/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - accuracy: 0.6627 - loss: 1.0186 - val_accuracy: 0.6687 - val_loss: 1.0001
Epoch 7/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.6630 - loss: 1.0050 - val_accuracy: 0.6701 - val_loss: 0.9907
Epoch 8/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.6648 - loss: 1.0037 - val_accuracy: 0.6708 - val_loss: 0.9828
Epoch 9/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.6633 - loss: 0.9931 - val_accuracy: 0.6731 - val_loss: 0.9766
Epoch 10/10
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 10s 7ms/step - accuracy: 0.6675 - loss: 0.9787 - val_accuracy: 0.6733 - val_loss: 0.9721
<keras.src.callbacks.history.History at 0x79583ec2bec0>
```

```
base_model.trainable = True
for layer in base_model.layers[:100]:
    layer.trainable = False

model.compile(optimizer=tf.keras.optimizers.Adam(1e-5), loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, validation_split=0.2)
```

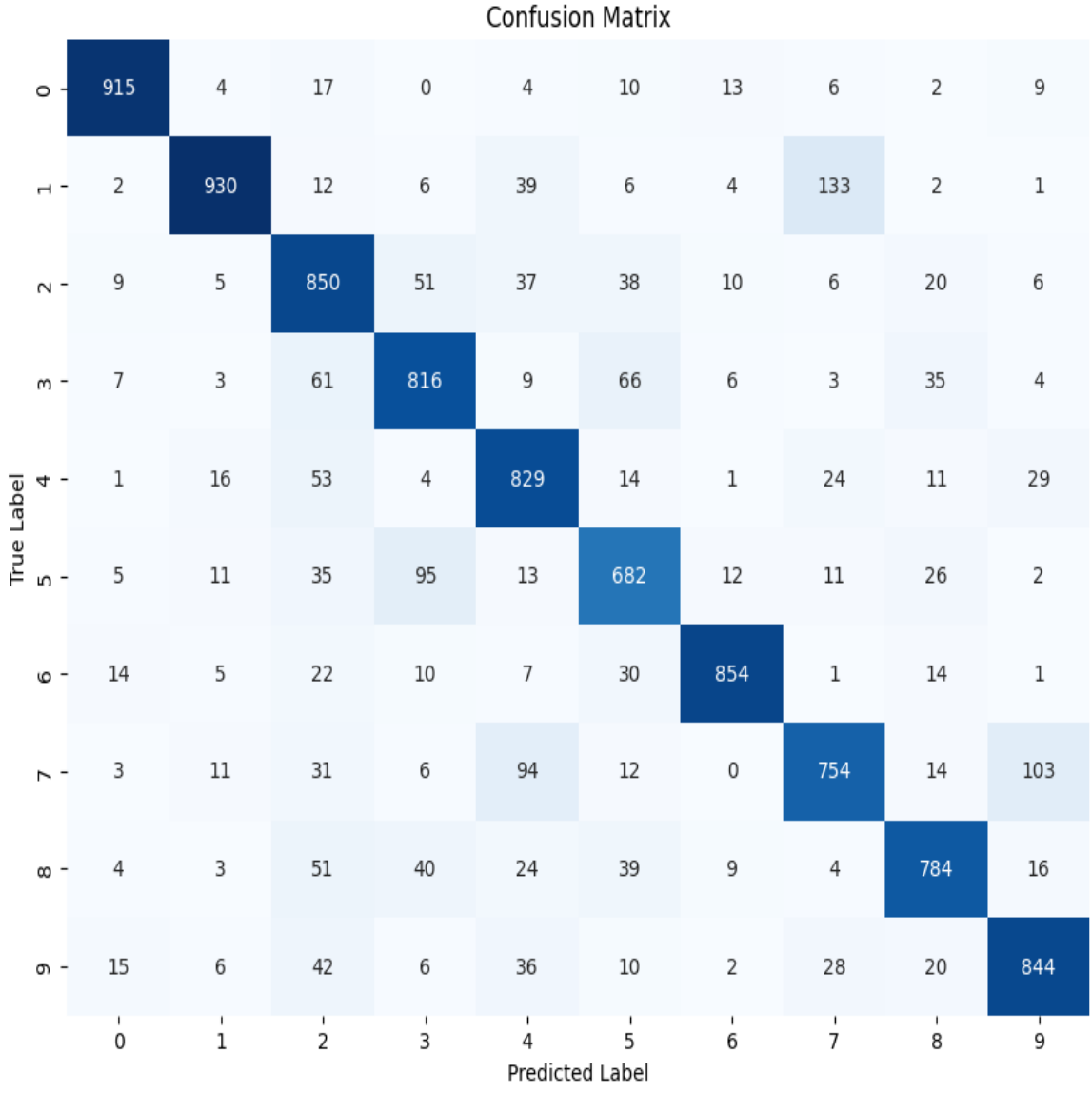
```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 36s 12ms/step - accuracy: 0.2263 - loss: 10.8758 - val_accuracy: 0.1616 - val_loss: 8.0069
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.4776 - loss: 2.5130 - val_accuracy: 0.2647 - val_loss: 2.3995
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.5971 - loss: 1.5561 - val_accuracy: 0.6446 - val_loss: 1.1514
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 14s 9ms/step - accuracy: 0.6941 - loss: 1.1050 - val_accuracy: 0.7961 - val_loss: 0.7486
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 13s 9ms/step - accuracy: 0.7578 - loss: 0.8581 - val_accuracy: 0.8398 - val_loss: 0.5935
<keras.src.callbacks.history.History at 0x79581a9cfb90>
```

```
loss, accuracy = model.evaluate(test_images, test_labels)
print(f"Test loss: {loss}")
print(f"Test accuracy: {accuracy}")
```

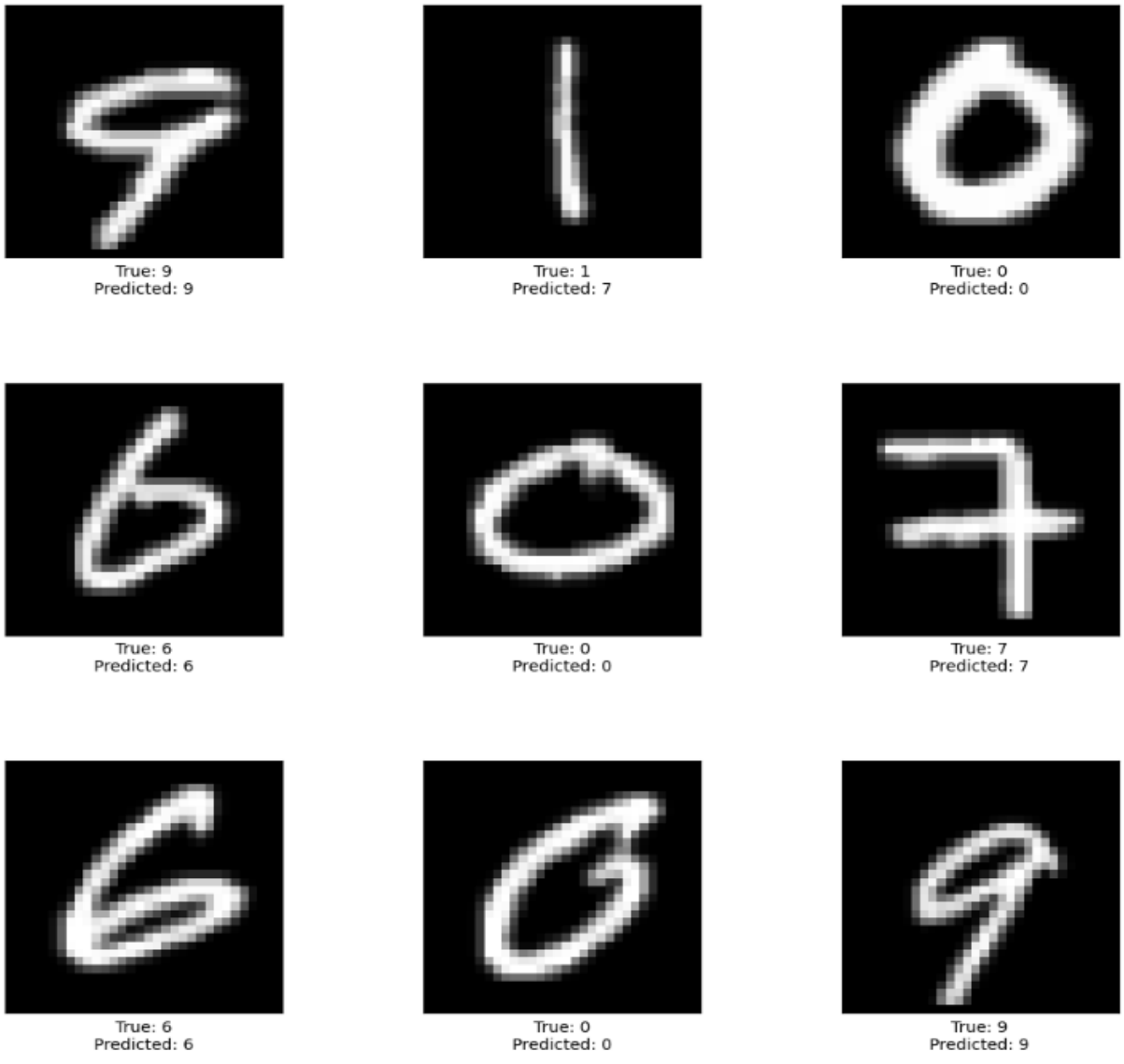
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 14s 35ms/step - accuracy: 0.8105 - loss: 0.6765
Test loss: 0.6357542276382446
Test accuracy: 0.8258000016212463
```

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

test_predictions = model.predict(test_images)
test_predictions_classes = np.argmax(test_predictions, axis=1)
test_true_classes = np.argmax(test_labels, axis=1)

cm = confusion_matrix(test_true_classes, test_predictions_classes)
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')
plt.show()
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 9s 16ms/step
```



```python
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()

test_images_gray = np.dot(test_images[...,:3], [0.2989, 0.5870, 0.1140])

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_sample(sample_images, sample_labels, sample_predictions)
```



Learning Outcome:

EXPERIMENT 10

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

Objective:

The objective of this experiment is to implement Transfer Learning using the pre-trained VGG16 model on an image dataset to leverage previously learned visual features for a new image classification task. The aim is to fine-tune the model for improved accuracy and reduced training time compared to building a model from scratch, while demonstrating the practical application of deep learning and feature extraction using a well-known convolutional neural network architecture.

Theory:

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.

Architecture:

a. Depth and 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.

Source 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()
```

Output:

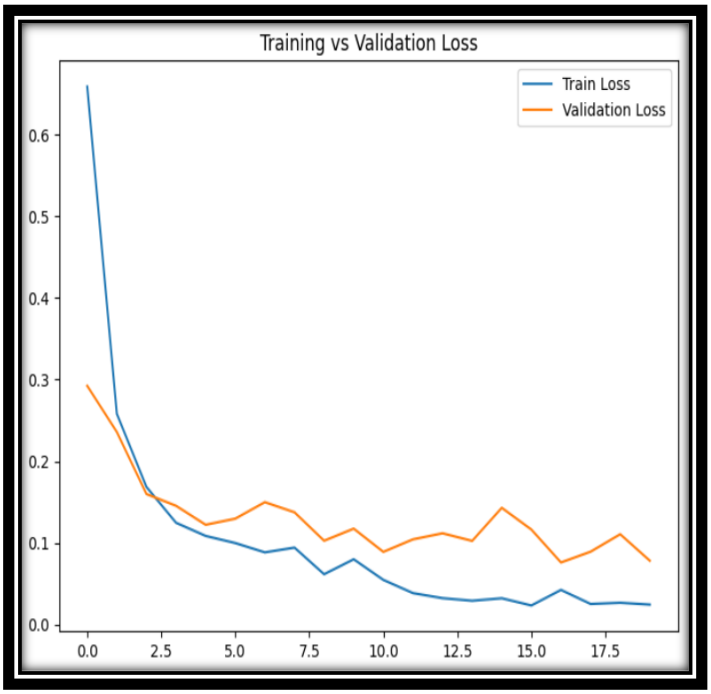
```
TensorFlow Version: 2.19.0
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

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 should call `super().__init__(**kwargs)` in its constructor.
  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 Outcome:

EXPERIMENT 11

Aim: NLP Analysis of Restaurant Reviews in Python.

Objective:

To perform Natural Language Processing (NLP) on restaurant reviews, analysing textual data to extract meaningful insights, identify sentiment patterns, and classify reviews as positive or negative, thereby demonstrating practical applications of text preprocessing, feature extraction, and machine learning techniques in Python.

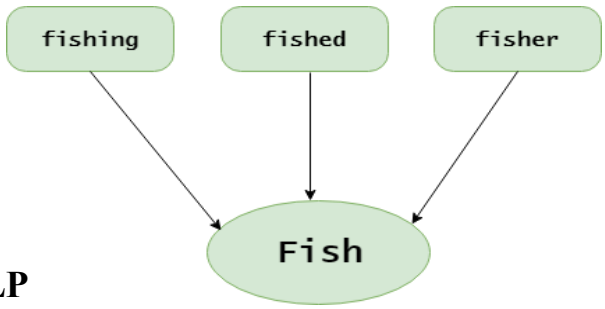
Theory:

Introduction to NLP

Natural Language Processing (NLP) is a significant branch of Artificial Intelligence (AI) that enables computers to understand, interpret, and generate human language in a valuable manner. NLP combines computational linguistics with machine learning and deep learning models to process large volumes of textual data. It allows systems to recognize patterns, extract information, and make decisions based on natural language inputs.

How NLP Works

NLP works through several stages, beginning with text preprocessing, which cleans and standardizes raw text. Common preprocessing steps include tokenization (splitting text into words or phrases), removal of stop-words (ignoring common words like “the,” “is,” “and”), and stemming/lemmatization (reducing words to their root form). Once the text is pre-processed, feature extraction techniques such as Bag of Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), or word embeddings are applied to convert text into numerical representations suitable for machine learning algorithms.



Applications of NLP

NLP has a wide array of applications across industries, including:

- Sentiment Analysis: Determining the positive or negative tone of text.
- Text Classification: Categorizing text into predefined labels.
- Chatbots and Virtual Assistants: Enabling conversational AI systems.
- Information Retrieval: Extracting relevant data from large textual corpora.
- Recommendation Systems: Suggesting products, services, or content based on textual reviews.

NLP in Restaurant Reviews

Applying NLP to restaurant reviews allows businesses to understand customer experiences and opinions at scale. Sentiment analysis on reviews helps identify:

- Customer satisfaction trends
- Common complaints or areas for improvement
- Positive aspects that drive customer loyalty

By classifying reviews as positive or negative, restaurants can make data-driven decisions, enhance services, and improve customer engagement. This approach provides actionable insights that are not feasible through manual review analysis.

Dataset Used

In this experiment, a dataset of restaurant reviews is used, consisting of textual feedback from customers along with their associated sentiment labels (positive or negative). Each review is pre-processed to remove noise and convert it into a structured format suitable for machine learning analysis. Feature extraction techniques like Bag of Words or TF-IDF are applied to prepare the data for classification models.

Importance of NLP in This Experiment

The experiment demonstrates the practical implementation of NLP concepts in Python, including text preprocessing, feature extraction, and sentiment classification. By analysing restaurant reviews, students can observe the power of NLP in understanding human language, automating insights extraction, and applying machine learning models for real-world textual data.

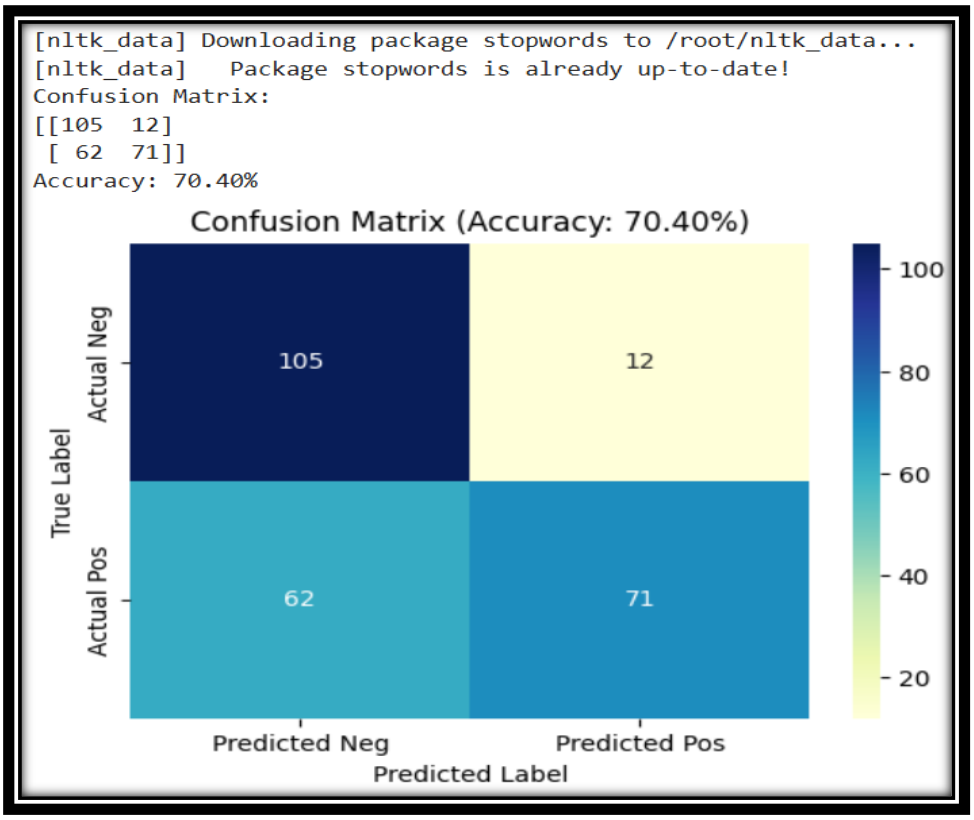
Source Code:

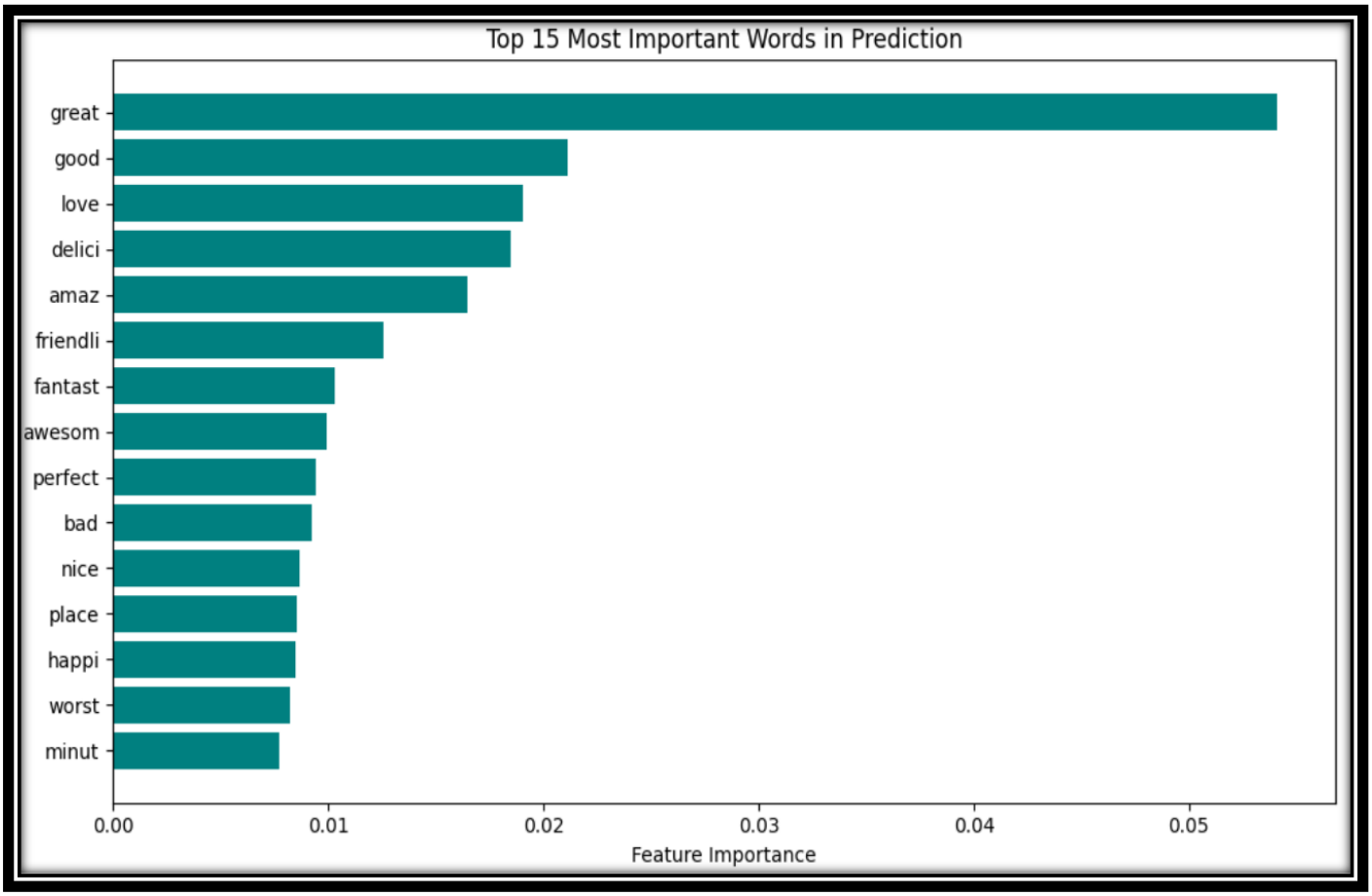
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
import nltk
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer

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

nltk.download('stopwords')

dataset = pd.read_csv('Restaurant_Reviews.tsv', delimiter='\t', quoting=3)  # quoting=3 to ignore quotes; adjust if needed
# The dataset has columns: "Review" and "Liked" (or similar) where 0 = negative, 1 = positive.

corpus = []
ps = PorterStemmer()
all_stopwords = set(stopwords.words('english'))

for i in range(len(dataset)):
    # Clean non-letters
    review = re.sub('[^a-zA-Z]', ' ', dataset['Review'][i])
    review = review.lower()
    review = review.split()
    # Remove stop-words, then stem
    review = [ps.stem(word) for word in review if word not in all_stopwords]
    review = ' '.join(review)
    corpus.append(review)
```

```
# Create Bag of Words (feature extraction)
cv = CountVectorizer(max_features=1500)  # you can tweak max_features
X = cv.fit_transform(corpus).toarray()
y = dataset.iloc[:, 1].values  # assuming second column is label (0/1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=0)

classifier = RandomForestClassifier(n_estimators = 501, criterion='entropy', random_state=0)
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
acc = accuracy_score(y_test, y_pred)

print("Confusion Matrix:")
print(cm)
print(f"Accuracy: {acc*100:.2f}%")

plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='YlGnBu',
            xticklabels=['Predicted Neg','Predicted Pos'],
            yticklabels=['Actual Neg','Actual Pos'])
plt.title(f'Confusion Matrix (Accuracy: {acc*100:.2f}%)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.show()

importances = classifier.feature_importances_
indices = np.argsort(importances)[-15:]  # Top 15 important words

plt.figure(figsize=(10,6))
plt.barh(range(len(indices)), importances[indices], color='teal')
plt.yticks(range(len(indices)), [cv.get_feature_names_out()[i] for i in indices])
plt.xlabel('Feature Importance')
plt.title('Top 15 Most Important Words in Prediction')
plt.tight_layout()
plt.show()
```

Output:

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
Confusion Matrix:
[[105  12]
 [ 62  71]]
Accuracy: 70.40%
```





Learning Outcome:

EXPERIMENT 12

Aim: Building an NLP model for Spam Detection using TFIDF (Term Frequency Inverse Document Frequency Vectorizer).

Objective:

To build a Natural Language Processing (NLP) model for detecting spam messages by converting text into numerical features using the TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer, and applying machine learning algorithms to accurately classify messages as spam or ham.

Theory:

Introduction to Spam Detection

Spam detection is one of the practical applications of NLP: the aim is to automatically classify messages (emails or SMS) into categories such as spam (unwanted, unsolicited, malicious) or ham (legitimate). By applying NLP methods, we can convert unstructured text into numerical features, and then train a model to distinguish between the two classes.

Why Use NLP for Spam Detection?

The volume of digital communication—emails, SMS, chat messages—has grown exponentially, and with it the amount of unwanted or malicious messages. Manual filtering is infeasible, hence the need for automated approaches. NLP offers the following advantages:

- It allows extraction of semantic or syntactic features from text (for example, keyword frequency, word patterns, unusual syntax)
- It enables scalable processing of large corpora of messages
- It supports deployment of machine learning or deep learning models that learn to generalize from examples, rather than relying solely on ad-hoc rule sets
  Applying NLP to spam detection thus improves accuracy, adaptability (to new spam patterns) and operational efficiency.

Workflow of Spam Detection Using NLP

The workflow for this experiment includes the following steps:

- Importing required libraries: e.g., pandas, numpy, nltk (for text processing), TensorFlow/Keras (for modelling) etc.
- Loading the dataset: Typically a labelled dataset where each message or email is annotated as spam or ham. For example: Emails.csv with columns for the message text and the label.
- Data preprocessing:
  - Cleaning text (removing punctuation, converting to lowercase, stripping whitespace)

  - Tokenization (splitting text into words or tokens)
  - Stop-words removal (eliminating frequently occurring but semantically weak words such as “the”, “is”, “and”)
  - Stemming or lemmatization (reducing words to root form)
  - Optionally balancing the dataset if one class is over-represented (e.g., many more ham than spam) to avoid bias.

- Feature extraction / vectorization: One of the most widely used techniques is TF-IDF (Term Frequency–Inverse Document Frequency). This converts each message into a fixed-length numerical vector, reflecting how important each term is in the message relative to the corpus. Using TF-IDF helps identify discriminative words that often appear in spam more than in ham, or vice-versa.
- Model building: Once data is vectorized, you can feed it into a classification model. The referenced article demonstrates deep learning (TensorFlow/Keras) for spam classification.
- Model training and evaluation: Split the data into training and test (or validation) sets. Train the model, monitor metrics like accuracy, precision, recall, F1-score. Evaluate on unseen data.
- Deployment / use: After achieving acceptable performance, the model can be applied to incoming messages to automatically classify them as spam/ham and act appropriately (filtering, alerting, quarantine).

Dataset for Spam Detection

The dataset contains 5,171 rows and 4 columns in one instance. The label distribution is often imbalanced: typically far more ham than spam messages, hence, balancing techniques (under sampling majority class) may be applied.

Why Use TF-IDF in This Experiment

TF-IDF provides a robust and interpretable way to convert text into numerical representation:

- Term Frequency (TF) quantifies how frequently a word occurs in a document (message).
- Inverse Document Frequency (IDF) penalizes words which are very common across the corpus (and thus less discriminative).
  Thus TF-IDF emphasises words that are both frequent in a given message and relatively rare across the corpus, making them potentially predictive of class (spam vs ham).
  Using TF-IDF in spam detection helps the model focus on relevant terms (e.g., “lottery”, “win”, “click”, “urgent”, etc) while down-weighting generic words (e.g., “please”, “thank”, “hello”).
  In the context of your experiment, applying TF-IDF vectorizer will allow you to:
- Efficiently transform many text messages into vectors

- Train a machine learning classifier on the vectors (e.g., logistic regression, SVM, or deep learning)

Applications and Significance

Spam detection via NLP and TF-IDF vectorization has practical significance:

- Email service providers filter out unsolicited or malicious messages before user inboxes.
- Telecom providers detect spam SMS and block or flag them.
- Organizations protect users and systems from phishing, malware, unwanted advertising.

Source Code:

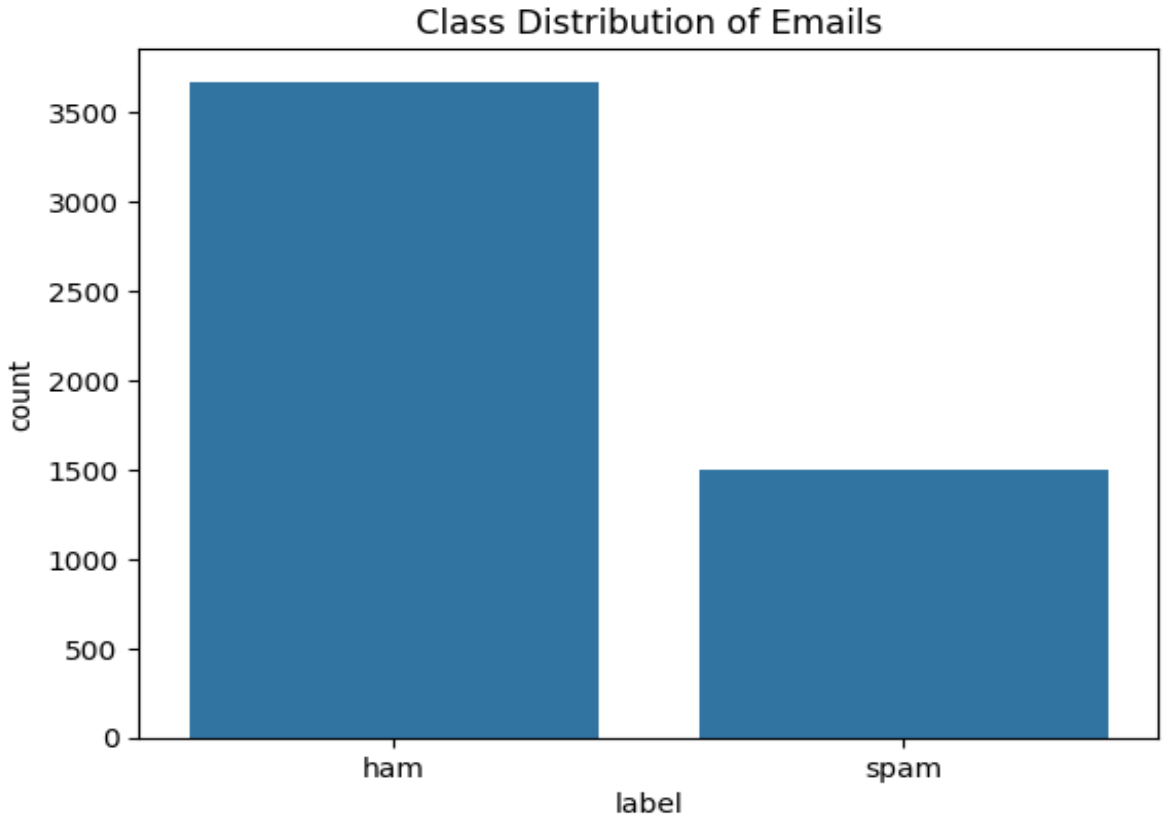
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import string
import nltk
from nltk.corpus import stopwords
from wordcloud import WordCloud
nltk.download('stopwords')

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')
data = pd.read_csv('spam_ham_dataset.csv')
data.head()
data.shape
sns.countplot(x='label', data=data)
plt.title("Class Distribution of Emails")
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)

balanced_data = pd.concat([ham_msg_balanced, spam_msg]).reset_index(drop=True)

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()
```

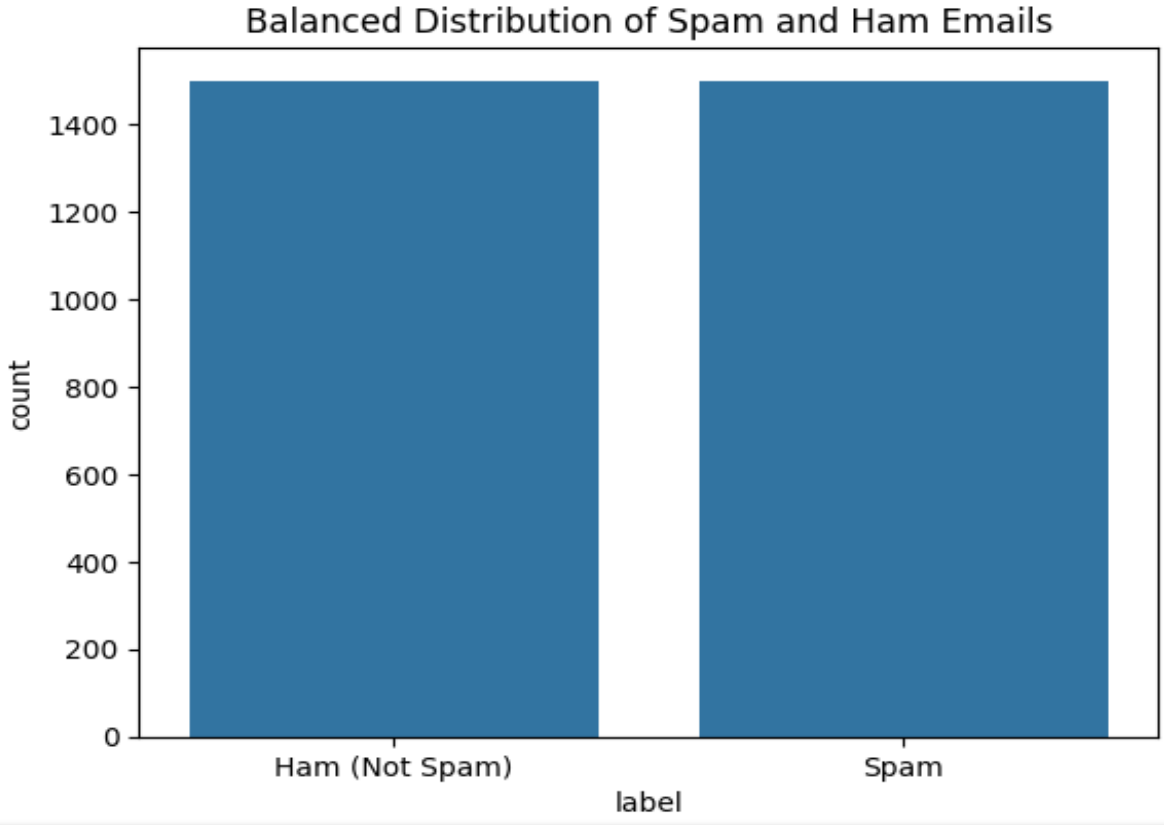
```
# Clean the text
balanced_data['text'] = balanced_data['text'].str.replace('Subject', '')
balanced_data.head()

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()

# Helper function to remove the stop words.
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()

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')
```

```
# Tokenization and Padding
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
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')])

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

model.summary()
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])

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

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()
```

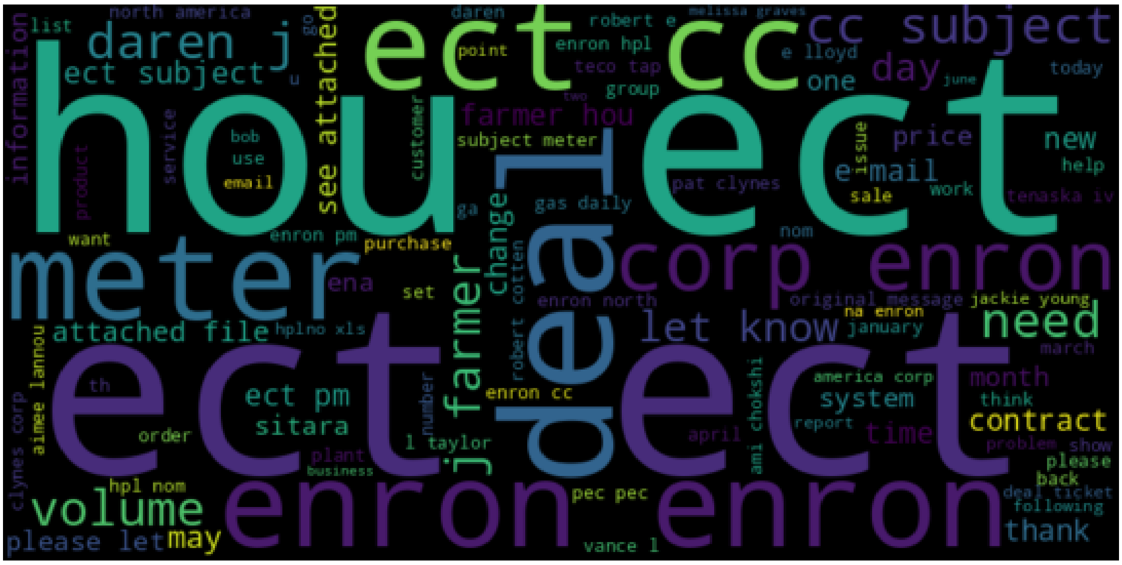
Output:

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





WordCloud for Non-Spam Emails



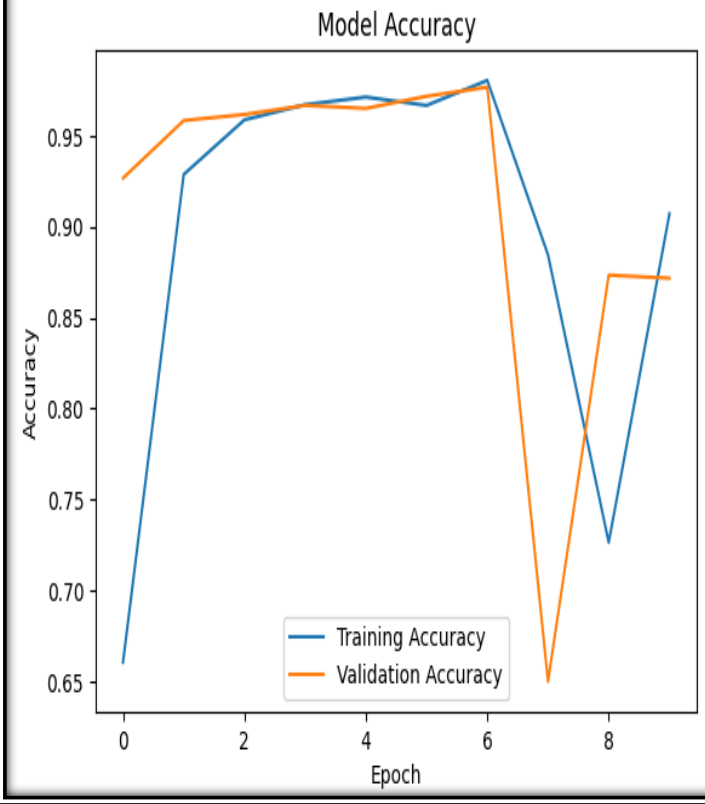
WordCloud for Spam Emails

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)

```
Epoch 1/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 8s 62ms/step - accuracy: 0.5436 - loss: 0.6714 - val_accuracy: 0.9267 - val_loss: 0.2763 - learning_rate: 0.0010
Epoch 2/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 4s 48ms/step - accuracy: 0.9221 - loss: 0.2586 - val_accuracy: 0.9583 - val_loss: 0.1736 - learning_rate: 0.0010
Epoch 3/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 3s 46ms/step - accuracy: 0.9611 - loss: 0.1623 - val_accuracy: 0.9617 - val_loss: 0.1585 - learning_rate: 0.0010
Epoch 4/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 3s 46ms/step - accuracy: 0.9635 - loss: 0.1423 - val_accuracy: 0.9667 - val_loss: 0.1395 - learning_rate: 0.0010
Epoch 5/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 5s 60ms/step - accuracy: 0.9673 - loss: 0.1345 - val_accuracy: 0.9650 - val_loss: 0.1558 - learning_rate: 0.0010
Epoch 6/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 3s 44ms/step - accuracy: 0.9731 - loss: 0.1151 - val_accuracy: 0.9717 - val_loss: 0.1190 - learning_rate: 0.0010
Epoch 7/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 5s 47ms/step - accuracy: 0.9788 - loss: 0.0925 - val_accuracy: 0.9767 - val_loss: 0.1109 - learning_rate: 0.0010
Epoch 8/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 4s 57ms/step - accuracy: 0.9681 - loss: 0.1325 - val_accuracy: 0.6500 - val_loss: 0.6097 - learning_rate: 0.0010
Epoch 9/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 4s 44ms/step - accuracy: 0.6767 - loss: 0.5501 - val_accuracy: 0.8733 - val_loss: 0.4220 - learning_rate: 0.0010
Epoch 10/20
75/75 ━━━━━━━━━━━━━━━━━━━━ 6s 51ms/step - accuracy: 0.9122 - loss: 0.3240 - val_accuracy: 0.8717 - val_loss: 0.3740 - learning_rate: 5.0000e-04
19/19 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9820 - loss: 0.0871
Test Loss : 0.11091102659702301
Test Accuracy : 0.9766666889190674
```



Learning Outcome: