RIYA GOEL 35517711623

Experiment-1

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

OBJECTIVES:

1.Grasp the foundational difference between TensorFlow, the core library, and Keras, the simplified API built on top of it.

2.Learn the three-step process for creating a neural network with Keras: model building, compiling, and training

THEORY:

TENSERFLOW

Introduction:

TensorFlow is an open-source machine learning and deep learning framework developed by the Google Brain Team in 2015. It provides tools to build, train, and deploy AI models for tasks like image recognition, natural language processing, recommendation systems, and more.

It works across multiple platforms (desktop, mobile, web, and cloud) , supports multiple programming languages (Python, C++, JavaScript, Swift) and runs on CPUs, GPUs, and TPUs for high performance.TensorFlow is used by companies like Google, Airbnb, Uber, Tesla, and healthcare organizations for real-time, large-scale AI solutions.

Unlike simpler libraries, TensorFlow is end-to-end — meaning it covers the full ML workflow:

- Data loading & preprocessing
- Model creation
- Training & evaluation
- Deployment to web, mobile, and edge devices

Key Features:

- **End-to-End ML Platform** – Covers the entire pipeline from data preprocessing to deployment.
- **Multi-Language Support** – Works with Python, C++, JavaScript (TF.js), Swift, and Java.
- Cross-Platform Compatibility – Runs on Windows, macOS, Linux, Android, iOS, and web browsers.
- **Acceleration** – Supports GPU and TPU for faster training.
- **Flexible Execution** – Eager execution for immediate results and graph mode for optimization.
- **Large Community & Support** – Backed by Google with extensive documentation and community resources.

RIYA GOEL 35517711623

Core Concepts:

1.**Tensors:** The fundamental data structure is the tensor, a multi-dimensional array for all computations.

2.**Computational Graph:** Operations are organized into a graph, enabling efficient parallel processing.

3.**Variables:** Special tensors that hold and update a model's trainable parameters.

4.**Ecosystem:** A comprehensive suite of tools for everything from building models to deployment.

5.**Scalability:** Designed to run on diverse hardware, from a single CPU to a vast network of GPUs.

Case Studies:

1. Airbnb – Price Optimization Using TensorFlow:

The Challenge:

Airbnb needed a scalable way to optimize rental pricing for millions of listings worldwide. Static pricing models or manual adjustments led to missed revenue opportunities and inconsistent pricing experiences. **The Solution:**

Airbnb uses TensorFlow to build machine learning models that predict optimal listing prices. These models take into account seasonality, demand fluctuations, competitor pricing, booking history, and listing features. The deep learning framework enables rapid retraining of models as market conditions change. As a result, hosts maximize revenue while guests are offered competitive prices.

2. Coca-Cola – Smart Vending Machines with TensorFlow The Challenge:

Coca-Cola wanted to modernize its vending machines to provide personalized product recommendations and optimize inventory management. Traditional vending systems lacked intelligence and real-time responsiveness.

The Solution:

Using TensorFlow, Coca-Cola integrated deep learning models into its IoT-enabled vending machines. These models analyze customer interactions, local preferences, and purchase history to make product recommendations. TensorFlow also helps forecast demand for specific products, improving restocking efficiency. This has led to increased sales and reduced operational costs.

3.Twitter – Hate Speech and Spam Detection Using TensorFlow

The Challenge:

Twitter needed to detect harmful content like hate speech and spam at scale. Traditional filters struggled with nuance and volume.

The Solution:

Twitter uses TensorFlow to train deep learning models on large datasets of labeled tweets. These models analyze the language, patterns, and context of tweets, improving over time through user reports and moderation feedback. This allows Twitter to detect abuse in real time and reduce reliance on manual moderation.

KERAS:

RIYA GOEL 35517711623

Introduction:

Keras is an open-source high-level deep learning API developed by François Chollet in 2015. It is designed to be simple, user-friendly, and modular, making it easier to build and train neural networks quickly.

Originally, Keras could run on multiple backends like TensorFlow, Theano, and CNTK. Since 2019, it has been officially integrated into TensorFlow as tf.keras, making it the default high-level API for TensorFlow.

Keras is widely used in research, education, and industry because it allows developers to create AI models with minimal code and without deep knowledge of low-level operations.

Key Features:

- **High-Level API** – Simple and readable code for quick model building.
- **Runs on Multiple Backends** – Originally worked with TensorFlow, Theano, CNTK (now mostly with TensorFlow).
- **User-Friendly** – Focuses on developer productivity with minimal boilerplate code.
- **Modular Design** – Models are made up of interchangeable building blocks like layers, optimizers, and loss functions.
- **Pretrained Models** – Built-in models for transfer learning.
- **Cross-Platform** – Compatible with Windows, macOS, Linux, and cloud platforms.
- **Data Handling:** Keras is compatible with a variety of data formats, including NumPy arrays, Pandas DataFrames, and native TensorFlow tf.data.Dataset objects, making data preprocessing and training straightforward.

Core Concepts:

1.Models: The central data structure in Keras is the Model. The two most common types ar

Sequential API: A linear stack of layers, suitable for simple, feedforward neural networks.

2. Functional API: Allows for more complex architectures with multiple inputs, outputs, and shared layers.

3.Layers: Layers are the fundamental building blocks of a neural network. Keras provides a wide variety of pre-built layers, including Dense (for fully connected networks), Conv2D (for convolutional networks), and LSTM (for recurrent networks).

4.Compilation: Before training, a model must be "compiled." This is where you configure the learning process by specifying:

5.Optimizer: The algorithm used to update the model's weights.

6.Loss Function: A function that measures the model's performance and guides the optimizer.

7. Metrics: A list of metrics to monitor during training and testing, such as accuracy.

Case Studies:

RIYA GOEL 35517711623

1. Netflix – Content Recommendation Using Keras

The Challenge:
Netflix needed to deliver highly personalized recommendations to a global user base. Traditional collaborative filtering methods couldn't capture deeper user preferences across genres, languages, and contexts.

The Solution:
Netflix uses Keras to build deep learning recommendation systems that learn user behavior patterns over time. These models analyze viewing history, content metadata, and user interactions. Keras's simplicity and modularity allow rapid experimentation and deployment, enabling Netflix to continuously improve content suggestions and boost viewer engagement.

2. Uber – ETA Prediction with Keras

The Challenge:
Uber required highly accurate Estimated Time of Arrival (ETA) predictions for millions of rides daily. Classical models lacked precision during dynamic traffic and weather conditions.

The Solution:
Uber leveraged Keras to create deep neural networks that process real-time GPS data, historical trip information, and traffic patterns. The models, trained and deployed using Keras's user-friendly interface, significantly improved ETA accuracy. This enhanced customer satisfaction and operational efficiency.

3. Healthcare – Early Disease Detection Using Keras

The Challenge:
Early detection of diseases from medical images (e.g., CT, MRI, X-ray) is critical, but manual interpretation is slow and subject to error.

The Solution:
Medical institutions use Keras to build convolutional neural networks (CNNs), often leveraging transfer learning, to analyze multi-modal imaging data. These models are trained to detect abnormalities—such as early-stage cancer or diabetic retinopathy—with over 95% accuracy.

Regression vs Classification with TensorFlow & Keras

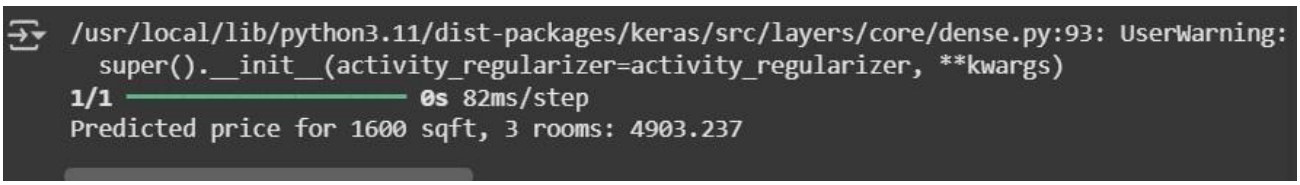
Regression models predict a continuous numerical value. The goal is to estimate a value within a certain range, rather than to classify it into a category. Think of a line of best fit in a scatter plot; the model tries to draw a line (or a more complex curve) that minimizes the distance to all the data points, allowing it to predict a new point's location on that line.

Classification models predict a discrete category or class label. The model's output is not a number on a continuous scale, but rather a choice from a predefined set of options. For example, a model might classify an email as "spam" or "not spam."

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

```
import pandas as pd import numpy
as np import tensorflow as tf from
tensorflow import keras from
tensorflow.keras import layers
df = pd.read_csv("simple_regression_dataset.csv") X
= df[["Size_sqft", "Rooms"]] y = df["Price"] model =
keras.Sequential([ layers.Dense(16,
activation="relu", input_shape=[2]), layers.Dense(8,
activation="relu"), layers.Dense(1)
])
model.compile(optimizer="adam", loss="mse", metrics=["mae"])
model.fit(X, y, epochs=200, verbose=0) new_data =
pd.DataFrame([[1600, 3]], columns=["Size_sqft", "Rooms"]) pred =
model.predict(new_data)
print("Predicted price for 1600 sqft, 3 rooms:", pred[0][0])
```

OUTPUT:



2.Code using Tensorflow and Keras for classification (pass/fail prediction):

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

df = pd.read_csv("details.csv")
df["Result"] = df["Result"].map({"Fail": 0, "Pass": 1})
X = df[["Study_Hours", "Attendance"]] y =
df["Result"]

model = keras.Sequential([     layers.Dense(16,
activation="relu", input_shape=[2]),     layers.Dense(8,
activation="relu"),     layers.Dense(1,
activation="sigmoid")
])

model.compile(optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"])
model.fit(X, y, epochs=200, verbose=0)

new_data = pd.DataFrame([[6, 85]], columns=["Study_Hours", "Attendance"])
pred = model.predict(new_data) pred_label = int(pred[0][0] > 0.5)
print("Predicted result (1=Pass, 0=Fail):", pred_label)
```

OUTPUT:

```
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:93: UserWarning:
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 64ms/step
Predicted result (1=Pass, 0=Fail): 1
```

LEARNING OUTCOME:

RIYA GOEL 35517711623

Experiment-2

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

Theory:

Artificial Neural Networks (ANNs) are computational models inspired by the human brain, designed to recognize complex patterns in data. They consist of layers of interconnected neurons that process inputs and adjust weights to minimize errors during training.

1. ANN for Regression:

Regression refers to predicting continuous numerical values (e.g., predicting house prices, stock prices, or temperature).

Model Structure:

- Sequential Model: Layers are arranged one after another in a pipeline.
- Input Layer: Takes the number of features (independent variables) from the dataset.
- Hidden Layer: A fully connected dense layer with ReLU activation to capture non-linear relationships.
- Output Layer: A single neuron without activation function, since regression predicts real values.

Compilation:

- Optimizer: Stochastic Gradient Descent (SGD), which updates weights step by step.
- Loss Function: Mean Squared Error (MSE), which penalizes large errors more strongly.
- Metric: Mean Absolute Error (MAE), which gives an interpretable error value in real units.

2. ANN for Classification

Classification is the process of predicting categories or classes (e.g., recognizing digits, spam detection, disease diagnosis).

Model Structure:

- Input Layer: Accepts features or image inputs (for example, a 28×28 image in MNIST).
- Flatten Layer: Converts multi-dimensional input (e.g., 2D images) into a single vector.
- Hidden Layers: Dense layers with ReLU activation to detect complex features and non-linear patterns.

- Output Layer:
  - For binary classification → one neuron with Sigmoid activation.
  - For multi-class classification → one neuron per class with Softmax activation.

Compilation:

- Optimizer: Adam (an adaptive optimizer widely used for classification).
- Loss Function:
  - Binary Crossentropy for binary classification.
  - Sparse Categorical Crossentropy for multi-class classification with integer labels.
- Metric: Accuracy, which measures the proportion of correct predictions.

Activation Functions in ANN:

Activation functions introduce non-linearity into the network, enabling the ANN to learn complex decision boundaries. Without them, even a deep network would behave like a simple linear model.

1. ReLU (Rectified Linear Unit)

- Formula: f(x) = max(0, x)
- Range: [0, ∞)
- Usage: Hidden layers (for both regression and classification).
- Advantages: Computationally efficient, reduces vanishing gradient problem, works well in deep models.
- Limitation: Some neurons may "die" if their output remains zero for all inputs.

2. Softmax

- Formula:

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

- Range: [0, 1], with outputs summing to 1.
- Usage: Output layer of multi-class classification models.
- Purpose: Converts raw scores into class probabilities.
- Example: If the output is [0.1, 0.7, 0.2], the model predicts the second class with 70% Confidence.

3. Sigmoid (Logistic Function)

- Formula:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Range: (0, 1)
- Usage: Output layer of binary classification models.
- Purpose: Maps output into a probability between 0 and 1.
- Limitation: Can cause vanishing gradient problems in deep networks.

Source Code:

Regression

```
[29] import tensorflow as tf
     housing = tf.keras.datasets.boston_housing
     (X_train, y_train), (X_test, y_test) = housing.load_data()
```

```
[30] model = tf.keras.models.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.summary()
```

```
Model: "sequential_4"
```

Layer (type)	Output Shape	Param #
dense_8 (Dense)	(None, 64)	896
dense_9 (Dense)	(None, 1)	65

```
Total params: 961 (3.75 KB)
Trainable params: 961 (3.75 KB)
Non-trainable params: 0 (0.00 B)
```

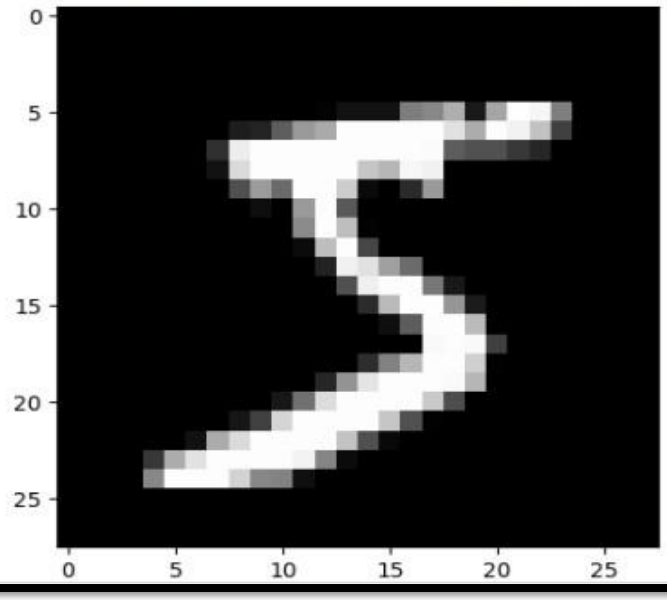
```
[31] history = model.fit(X_train, y_train, epochs=10)

Epoch 1/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 4271356933060952064.0000 - mae: 575089152.0000
Epoch 2/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 15913421841104896.0000 - mae: 125919880.0000
Epoch 3/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 9411124822802432.0000 - mae: 96835200.0000
Epoch 4/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 5565695110676480.0000 - mae: 74468432.0000
Epoch 5/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 3291526838878208.0000 - mae: 57267896.0000
Epoch 6/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 1946593800486912.0000 - mae: 44040292.0000
Epoch 7/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 1151206927892480.0000 - mae: 33867976.0000
Epoch 8/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - loss: 680818552864768.0000 - mae: 26045230.0000
Epoch 9/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 402633084895232.0000 - mae: 20029366.0000
Epoch 10/10
13/13 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 238115453140992.0000 - mae: 15403036.0000
```

```
[32] loss, mae = model.evaluate(X_test, y_test)

4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 159005058007040.0000 - mae: 12609720.0000

[33] print(f"Mean Absolute error: {mae}")

Mean Absolute error: 12609720.0

[34] print(f"Loss: {loss}")

Loss: 159005074784256.0

[35] y_preds = model.predict(X_test)

4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 15ms/step

[36] model.evaluate(X_test, y_test)

4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - loss: 159005058007040.0000 - mae: 12609720.0000
[159005074784256.0, 12609720.0]
```

Classification

```
import tensorflow as tf
import matplotlib.pyplot as plt
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.mnist.load_data()
plt.imshow(X_train[0], cmap="gray")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
<matplotlib.image.AxesImage at 0x7c9b508e2720>
```



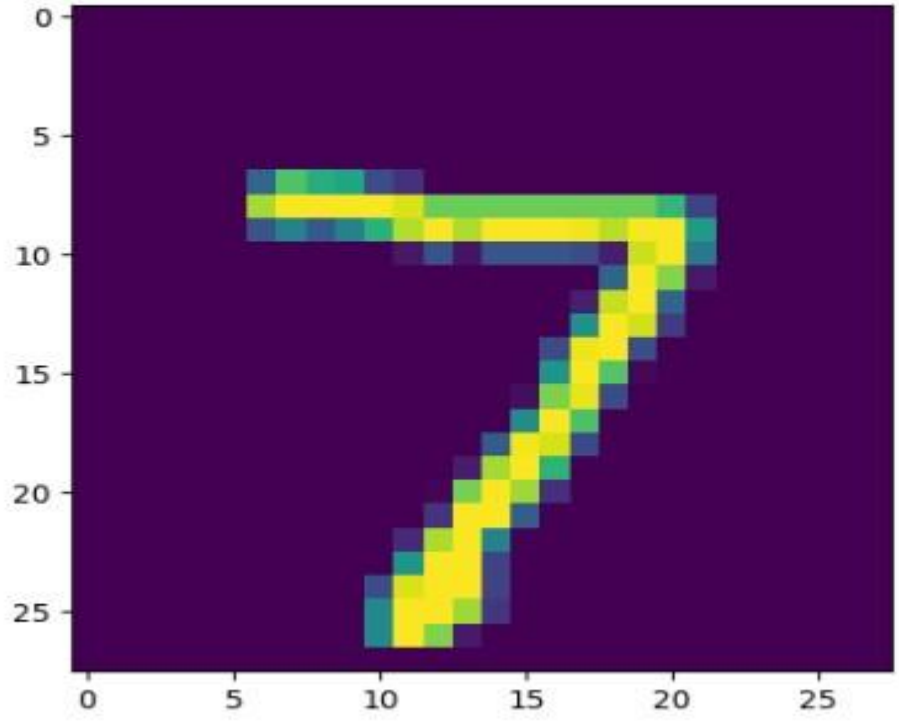

```
model = tf.keras.models.Sequential([
    tf.keras.layers.Input((28,28)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(28, activation='relu'),
    tf.keras.layers.Dense(28, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax'),
])
model.compile(optimizer='adam', loss=tf.keras.losses.SparseCategoricalCrossentropy(), metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=1)
```

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.5598 - loss: 3.5809
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8443 - loss: 0.5767
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8834 - loss: 0.4537
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8988 - loss: 0.3945
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9143 - loss: 0.3506
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9152 - loss: 0.3326
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9209 - loss: 0.3125
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9245 - loss: 0.2899
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9282 - loss: 0.2774
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9334 - loss: 0.2514
```

RIYA GOEL 35517711623

```
[39] model.evaluate(X_test, y_test)

313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9197 - loss: 0.3088
[0.27237075567245483, 0.9297999739646912]

[40] import numpy as np
     pred = model.predict(X_test[0].reshape(-1,28,28))

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 71ms/step

[42] pred

array([[6.3506922e-11, 2.8312434e-07, 5.5308146e-06, 1.6963122e-05,
        6.4058946e-07, 4.4978368e-08, 5.5671318e-12, 9.9979168e-01,
        7.8318720e-07, 1.8412074e-04]], dtype=float32)

[43] np.argmax(pred)

np.int64(7)
```

```
plt.imshow(X_test[0])

<matplotlib.image.AxesImage at 0x7c9b5055deb0>
```



Learning Outcome:

RIYA GOEL 35517711623

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.

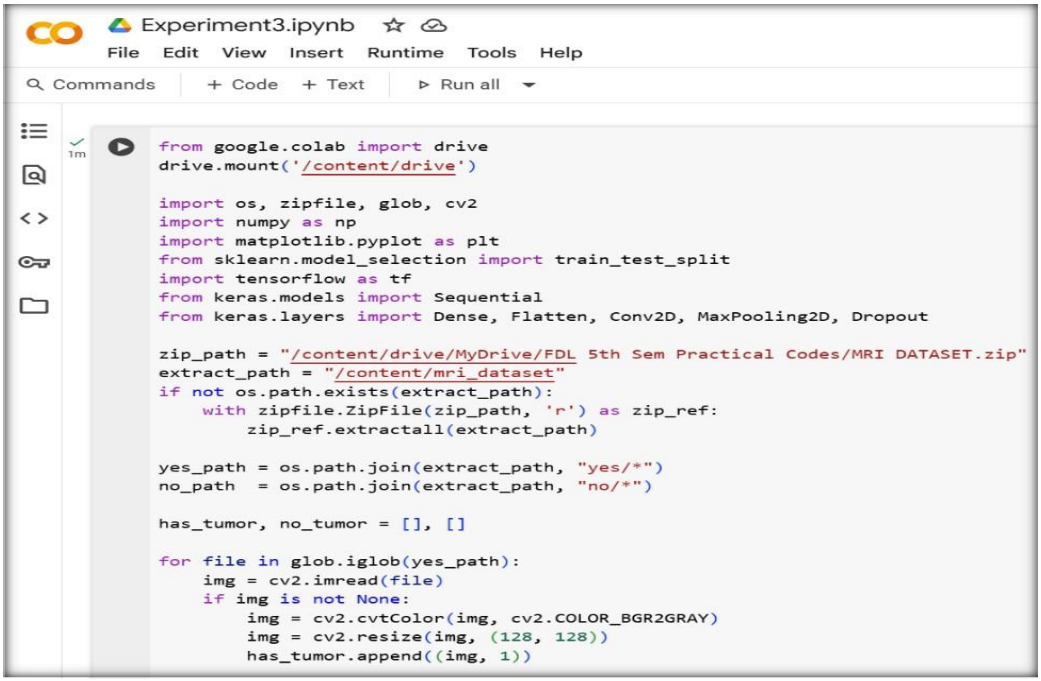
2.Reduces computational complexity compared to traditional fully connected networks.

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:



RIYA GOEL 35517711623

```
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()
```

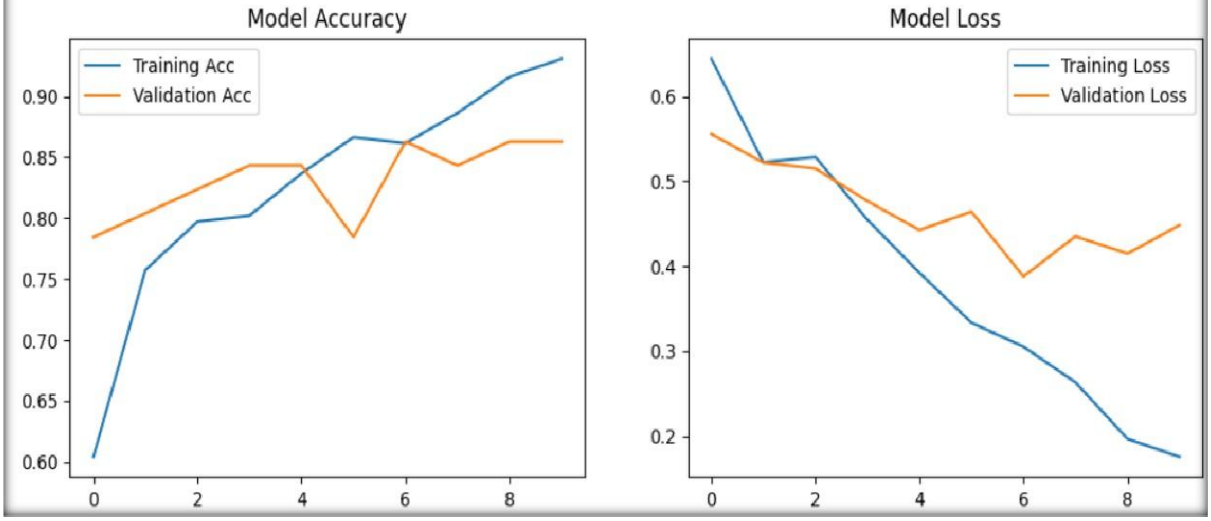
RIYA GOEL 35517711623

Outputs:

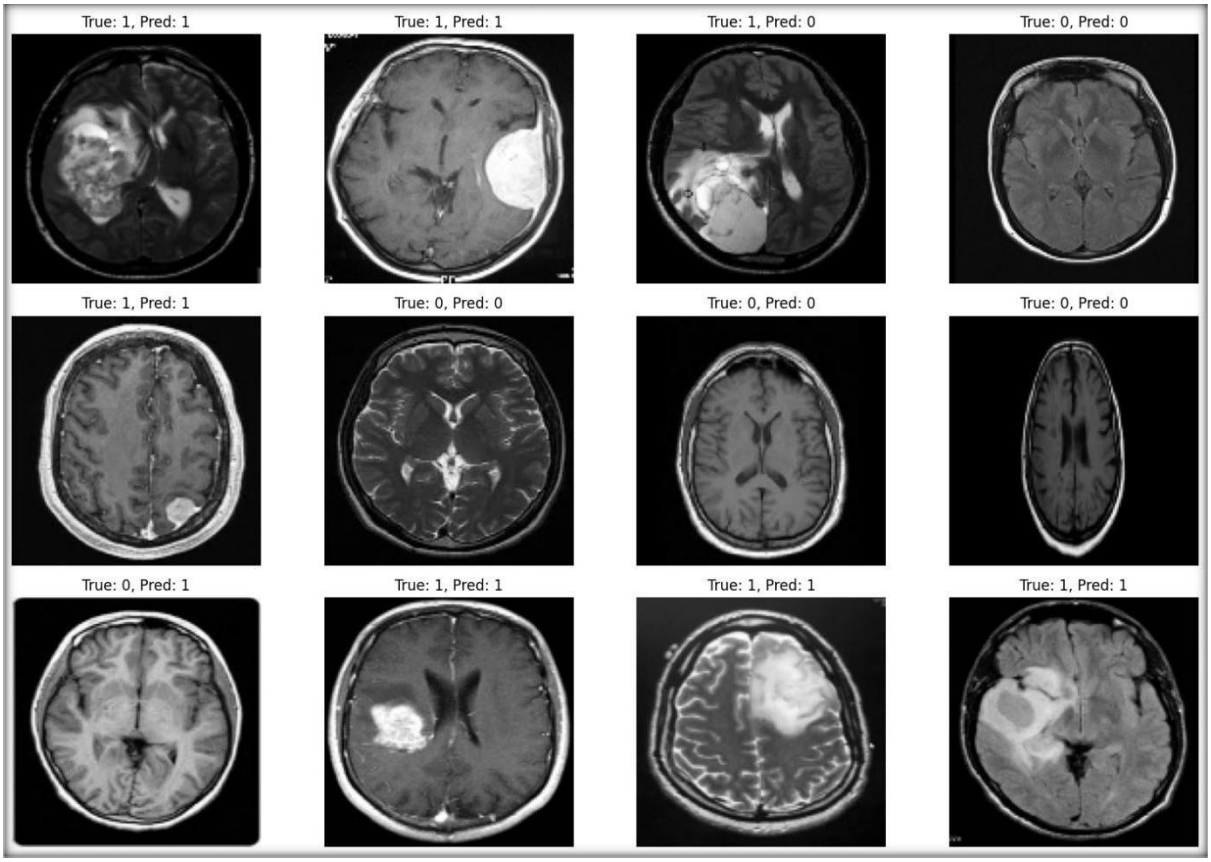
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
Model: "sequential_2"
```

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
```



RIYA GOEL 35517711623



Learning Outcome:

RIYA GOEL 35517711623

EXPERIMENT-4

Aim: Implementation of Autoencoders for dimensionality reduction in Python.

Objectives:

- To design and train an autoencoder model for dimensionality reduction using synthetic data.
- To evaluate the effectiveness of the autoencoder by comparing reconstruction loss and visualizing compressed representations.

Theory:

1. Architecture:

- **Encoder:** The encoder is the first component of the autoencoder, responsible for compressing the input data into a smaller and more meaningful representation. This compressed form is known as the *code* or *latent space*. It is achieved using layers with progressively fewer neurons, which capture the most important features of the input while discarding redundancy.
- **Decoder:** The decoder is the second component, which reconstructs the input data from the compressed latent representation. Its role is to ensure that the reduced information still retains enough detail to approximate the original data closely.

2. Loss Function:

- The loss function quantifies the difference between the original input and the reconstructed output. In the context of dimensionality reduction, minimizing this reconstruction error ensures that essential features are preserved while reducing data complexity.
- Commonly used loss functions include Mean Squared Error (MSE) for continuous data and Binary Crossentropy for binary or normalized data. The choice depends on the nature and scale of the dataset being processed.

3. Practical Implementation Steps:

- **Data Preparation:** Prepare the input data by applying standardization or normalization so that all features are on a comparable scale. This step is crucial for efficient learning and faster convergence.
- **Model Architecture:** Build an autoencoder using the Keras API. The architecture typically consists of an encoder (with a hidden layer for compression) and a decoder (to reconstruct the input from the latent space).
- **Loss Function and Optimizer:** Select a suitable loss function, such as MSE, and an optimizer like Adam, which is widely used due to its efficiency and adaptive learning rate.
- **Training:** Train the autoencoder on the prepared dataset, ensuring that the goal is to minimize reconstruction error. During training, the model learns compact feature representations while maintaining reconstruction accuracy.
- **Evaluation:** After training, evaluate the autoencoder's ability to reconstruct the input. This involves assessing the quality of reconstructions, checking the magnitude of reconstruction loss, and analyzing how effectively the dimensionality reduction captures relevant information.

Algorithm:

1.Generate Random Data: Generate a synthetic dataset (`data`) containing 1000 samples, each with 20 features.

2.Set Autoencoder Parameters: Define the input dimension (`input_dim`) based on the number of features in the dataset (20). Specify the encoding dimension (`encoding_dim`), determining the number of neurons in the hidden layer.

3.Build Autoencoder Model: Create an autoencoder model using the Keras Functional API. Define an input layer (`input_layer`) corresponding to the input dimension. Add a Dense layer (`encoder_layer`) with ReLU activation to perform dimensionality reduction. Add another Dense layer (`decoder_layer`) with a sigmoid activation to reconstruct the original input. Form the autoencoder model with the input and decoder layers.

4.Compile Autoencoder Model: Compile the autoencoder model using the Adam optimizer and mean squared error as the loss function.

5.Train Autoencoder: Train the autoencoder model using the synthetic data (`data`) for 50 epochs. Use mean squared error as the loss function. Shuffle the training data and allocate 20% for validation during training. Store training history for later visualization.

6.Plot Training Loss Over Epochs: Plot the training and validation loss over epochs using Matplotlib.

7.Extract Encoder Model: Create a separate model (`encoder`) that takes the input and outputs the encoder layer. Use this model to obtain the encoded representations of the input data.

8.Visualize Original and Encoded Data: Display original and encoded representations for a subset of the data (first 5 samples). Plot original data on the top row and encoded data on the bottom row.

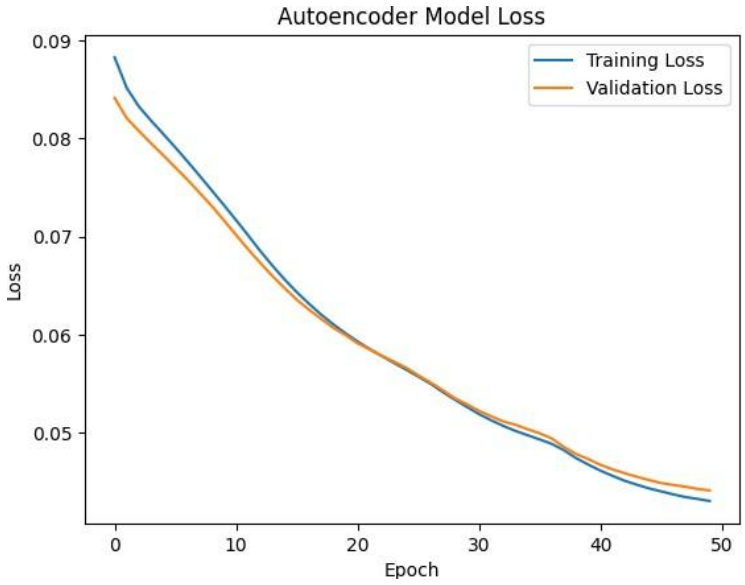
Source Code:

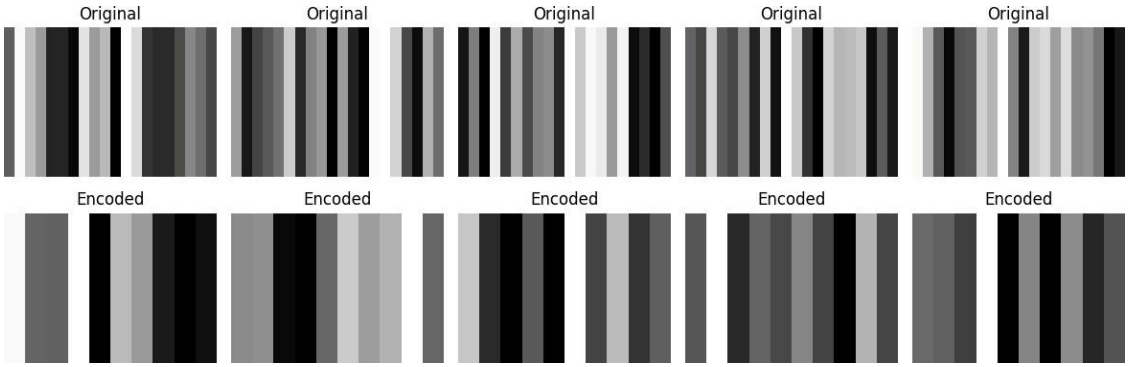
```
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Input, Dense
from keras.models import Model

np.random.seed(42)

data = np.random.rand(1000, 20)
input_dim = data.shape[1]
encoding_dim = 10

input_layer = Input(shape=(input_dim,))
encoder_layer = Dense(encoding_dim, activation='relu')(input_layer)
decoder_layer = Dense(input_dim, activation='sigmoid')(encoder_layer)

autoencoder = Model(inputs=input_layer, outputs=decoder_layer)
autoencoder.compile(optimizer='adam', loss='mean_squared_error')

history = autoencoder.fit(
    data, data,
    epochs=50,
    batch_size=32,
    shuffle=True,
    validation_split=0.2
)
```

```
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Autoencoder Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(loc='upper right')
plt.show()

encoder = Model(inputs=input_layer, outputs=encoder_layer)
encoded_data = encoder.predict(data)

n = 5
plt.figure(figsize=(12, 4))

for i in range(n):
    plt.subplot(2, n, i + 1)
    plt.imshow(data[i].reshape(1, -1), cmap='gray', aspect='auto')
    plt.title('Original')
    plt.axis('off')

    plt.subplot(2, n, i + 1 + n)
    plt.imshow(encoded_data[i].reshape(1, -1), cmap='gray', aspect='auto')
    plt.title('Encoded')
    plt.axis('off')

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

Output:

```
Epoch 1/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0902 - val_loss: 0.0841
Epoch 2/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0859 - val_loss: 0.0821
Epoch 3/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0838 - val_loss: 0.0808
Epoch 4/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0822 - val_loss: 0.0795
Epoch 5/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0810 - val_loss: 0.0783
Epoch 6/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0791 - val_loss: 0.0770
Epoch 7/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.0779 - val_loss: 0.0758
Epoch 8/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0758 - val_loss: 0.0745
Epoch 9/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0742 - val_loss: 0.0731
Epoch 10/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0732 - val_loss: 0.0717
Epoch 11/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0725 - val_loss: 0.0702
Epoch 12/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0709 - val_loss: 0.0687
Epoch 13/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0692 - val_loss: 0.0673
Epoch 14/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0673 - val_loss: 0.0660
Epoch 15/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0657 - val_loss: 0.0647
Epoch 16/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0639 - val_loss: 0.0636
Epoch 17/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0633 - val_loss: 0.0626
Epoch 18/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0626 - val_loss: 0.0616
Epoch 19/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0604 - val_loss: 0.0607
Epoch 20/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0601 - val_loss: 0.0600
Epoch 21/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - loss: 0.0589 - val_loss: 0.0591
Epoch 22/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0582 - val_loss: 0.0585
Epoch 23/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0583 - val_loss: 0.0578
Epoch 24/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0579 - val_loss: 0.0573
Epoch 25/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0570 - val_loss: 0.0566
Epoch 26/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0565 - val_loss: 0.0559
Epoch 27/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0556 - val_loss: 0.0551
Epoch 28/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0543 - val_loss: 0.0544
Epoch 29/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0536 - val_loss: 0.0535
Epoch 30/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0530 - val_loss: 0.0529
Epoch 31/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0518 - val_loss: 0.0522
Epoch 32/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0513 - val_loss: 0.0517
Epoch 33/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - loss: 0.0512 - val_loss: 0.0512
Epoch 34/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0502 - val_loss: 0.0508
Epoch 35/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0498 - val_loss: 0.0504
Epoch 36/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0488 - val_loss: 0.0500
Epoch 37/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0495 - val_loss: 0.0494
Epoch 38/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - loss: 0.0484 - val_loss: 0.0486
Epoch 39/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0467 - val_loss: 0.0478
Epoch 40/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0473 - val_loss: 0.0473
Epoch 41/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0464 - val_loss: 0.0467
Epoch 42/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0454 - val_loss: 0.0463
Epoch 43/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - loss: 0.0454 - val_loss: 0.0459
Epoch 44/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0450 - val_loss: 0.0455
Epoch 45/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0448 - val_loss: 0.0452
Epoch 46/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0439 - val_loss: 0.0449
Epoch 47/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0446 - val_loss: 0.0447
Epoch 48/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0437 - val_loss: 0.0445
Epoch 49/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0430 - val_loss: 0.0443
Epoch 50/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - loss: 0.0430 - val_loss: 0.0441
```





Learning Outcome:

RIYA GOEL 35517711623

EXPERIMENT-5

Aim: Application of Autoencoders on Image Dataset in Python.

Objectives:

- To design and implement an autoencoder/CAE model for image reconstruction and dimensionality reduction.
- To demonstrate practical applications of autoencoders such as denoising, anomaly detection, and effective feature extraction.

Theory:

1.Image Reconstruction

- **Encoder:** Autoencoders compress high-dimensional image data into a lower-dimensional representation in the latent space.
- **Decoder:** The compressed representation is decoded to reconstruct the original image.

2. Applications

- **Dimensionality Reduction:** Autoencoders capture essential features, reducing dimensionality while preserving critical information.
- **Image Denoising:** By training on noisy images and reconstructing clean images, autoencoders can remove noise effectively.
- **Anomaly Detection:** Autoencoders learn normal data patterns and identify anomalies by detecting deviations from the learned representations.

3. Convolutional Autoencoders (CAE)

- **Convolutional Layers:** Used in both encoder and decoder to capture spatial hierarchies in images for effective feature extraction.
- **Pooling Layers:** Employed in the encoder for down-sampling, reducing spatial dimensions while retaining key information.
- **Upsampling Layers:** Used in the decoder to reconstruct the original image dimensions from the latent space.

Algorithm :

1.Import Libraries: Import necessary libraries such as NumPy, Matplotlib, and relevant Keras modules.

2.Generate Random Data: Create a random dataset (data) with 1000 samples, each having 20 features.

3.Set Autoencoder Parameters: o Input dimension (input_dim) = 20 (number of features).

- Encoding dimension (encoding_dim) = 10 (neurons in hidden layer).

4.Build Autoencoder Model: o Define an input layer (input_layer). o Add a Dense layer (encoder_layer) with ReLU activation for dimensionality reduction. o Add a Dense layer (decoder_layer) with sigmoid activation to reconstruct the input.

- Form the autoencoder model using the input and decoder layers.

5.Compile Autoencoder Model: Use Adam optimizer and Mean Squared Error (MSE) as the loss function.

6.Autoencoder: Train on the dataset for 50 epochs, batch size = 32, with 20% validation data.
Store training history for visualization.

7.Plot Training Loss: Plot training and validation loss over epochs using Matplotlib.

8.Extract Encoder Model: Create a separate encoder model to output encoded (compressed) representations from the input.

9.Visualize Data: Display the original and encoded representations for the first 5 samples. Plot original data on the top row and encoded data on the bottom row.

Source Code:

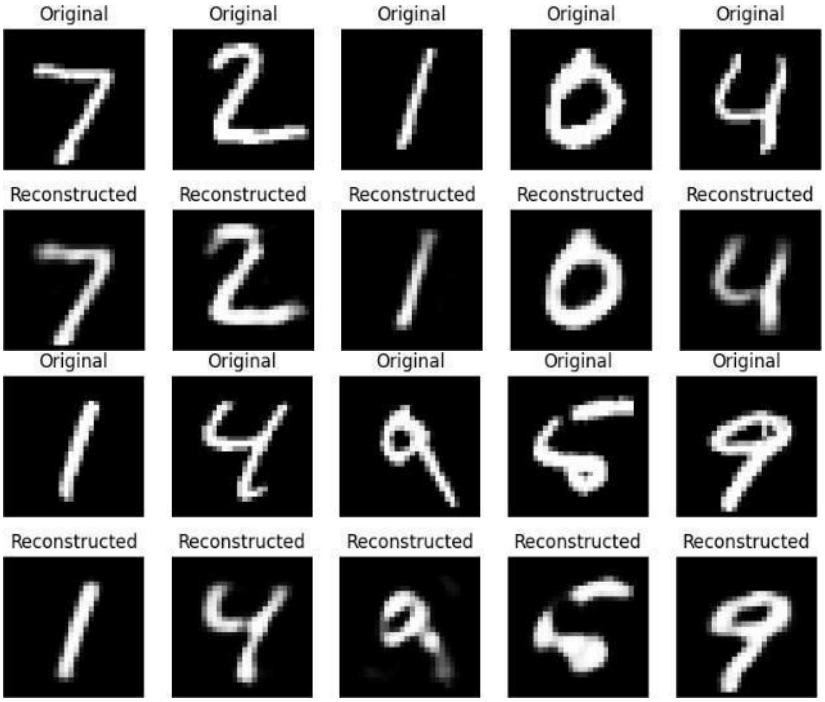
```
import numpy as np
import matplotlib.pyplot as plt
from keras.layers import Input, Dense
from keras.models import Model
from keras.datasets import mnist

(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_flat = x_train.reshape((len(x_train), np.prod(x_train.shape[1:])))
x_test_flat = x_test.reshape((len(x_test), np.prod(x_test.shape[1:])))

input_dim = 784
encoding_dim = 32

input_layer = Input(shape=(input_dim,))
encoder_layer = Dense(encoding_dim, activation='relu')(input_layer)
decoder_layer = Dense(input_dim, activation='sigmoid')(encoder_layer)

autoencoder = Model(inputs=input_layer, outputs=decoder_layer)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

```
autoencoder.fit(
    x_train_flat, x_train_flat,
    epochs=50,
    batch_size=256,
    shuffle=True,
    validation_data=(x_test_flat, x_test_flat)
)

decoded_imgs = autoencoder.predict(x_test_flat)

n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i], cmap='gray')
    plt.title('Original')
    plt.axis('off')

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

plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-data
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 1s 0us/step
Epoch 1/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 10ms/step - loss: 0.3848 - val_loss: 0.1892
Epoch 2/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1800 - val_loss: 0.1527
Epoch 3/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1480 - val_loss: 0.1329
Epoch 4/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - loss: 0.1311 - val_loss: 0.1213
Epoch 5/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1205 - val_loss: 0.1130
Epoch 6/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1123 - val_loss: 0.1067
Epoch 7/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1067 - val_loss: 0.1021
Epoch 8/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.1022 - val_loss: 0.0988
Epoch 9/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - loss: 0.0993 - val_loss: 0.0970
Epoch 10/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0973 - val_loss: 0.0951
Epoch 11/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0963 - val_loss: 0.0942
Epoch 12/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0954 - val_loss: 0.0937
Epoch 13/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0948 - val_loss: 0.0932
Epoch 14/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0943 - val_loss: 0.0930
Epoch 15/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0941 - val_loss: 0.0927
Epoch 16/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 18ms/step - loss: 0.0939 - val_loss: 0.0925
Epoch 17/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 6s 21ms/step - loss: 0.0938 - val_loss: 0.0924
Epoch 18/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 13ms/step - loss: 0.0936 - val_loss: 0.0923
Epoch 19/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0934 - val_loss: 0.0922
Epoch 20/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0933 - val_loss: 0.0921
Epoch 21/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0930 - val_loss: 0.0921
Epoch 22/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0932 - val_loss: 0.0920
Epoch 23/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0931 - val_loss: 0.0919
Epoch 24/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0932 - val_loss: 0.0919
Epoch 25/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0931 - val_loss: 0.0919
```

```
Epoch 26/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0930 - val_loss: 0.0918
Epoch 27/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 0.0931 - val_loss: 0.0918
Epoch 28/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0930 - val_loss: 0.0918
Epoch 29/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0929 - val_loss: 0.0918
Epoch 30/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0930 - val_loss: 0.0917
Epoch 31/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 14ms/step - loss: 0.0929 - val_loss: 0.0917
Epoch 32/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0928 - val_loss: 0.0917
Epoch 33/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0928 - val_loss: 0.0916
Epoch 34/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 35/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step - loss: 0.0929 - val_loss: 0.0917
Epoch 36/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0928 - val_loss: 0.0916
Epoch 37/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 38/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0929 - val_loss: 0.0916
Epoch 39/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 0.0930 - val_loss: 0.0916
Epoch 40/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 41/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 12ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 42/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - loss: 0.0925 - val_loss: 0.0915
Epoch 43/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 44/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0927 - val_loss: 0.0916
Epoch 45/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 46/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.0926 - val_loss: 0.0915
Epoch 47/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0927 - val_loss: 0.0915
Epoch 48/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0928 - val_loss: 0.0915
Epoch 49/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0928 - val_loss: 0.0915
Epoch 50/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.0926 - val_loss: 0.0915
313/313 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step
```



Learning Outcome:

RIYA GOEL 35517711623

EXPERIMENT-6

Aim: Improving Autocoder's Performance using convolution layers in Python (MNIST Dataset to be utilized).

Objective: To design and implement an autoencoder model using convolutional layers in Python and analyze its performance on the MNIST dataset, demonstrating how convolution-based architectures improve the quality of reconstructed images compared to traditional dense autoencoders.

Theory:

Introduction

Autoencoders are a class of unsupervised neural networks that aim to learn efficient representations of input data. They consist of two parts: an encoder, which compresses the input into a lower-dimensional latent representation, and a decoder, which reconstructs the input from this compressed form. The network is trained to minimize the reconstruction error between the original input and its output.

Traditional autoencoders are built using fully connected layers, which work reasonably well for low-dimensional data but are not ideal for image data. This is because dense layers ignore spatial locality in images, leading to blurred or less meaningful reconstructions.

Convolutional Autoencoders

Convolutional autoencoders (CAEs) address the limitations of traditional autoencoders by introducing convolutional and pooling layers in the encoder, and convolutional and upsampling layers in the decoder.

Encoder: The encoder applies convolutional filters to extract local features such as edges and textures. Pooling layers reduce spatial dimensions while retaining essential information, creating a compact and meaningful representation of the input.

Decoder: The decoder gradually reconstructs the image using upsampling and convolutional layers. Upsampling layers restore the spatial dimensions, while convolutional filters add detail and refine the reconstruction. A sigmoid activation function is often applied in the final layer to ensure the output values are in the range [0, 1], matching the normalized input image.

Advantages of Convolutional Autoencoders

- Preserve spatial features of images.
- Require fewer parameters compared to dense autoencoders.
- Learn hierarchical representations, from simple edges to complex shapes.
- Produce sharper and clearer reconstructions.

Dataset:

The MNIST dataset is used for this experiment. It consists of 70,000 grayscale images of handwritten digits (0–9), each with a size of 28×28 pixels. The dataset is divided into 60,000 training images and 10,000 testing images. All images are normalized by scaling pixel values between 0 and 1 and reshaped to fit the input requirements of the convolutional autoencoder.

Model Implementation:
The convolutional autoencoder is implemented in Python using TensorFlow/Keras.

- **Encoder:** Multiple convolutional layers with ReLU or LeakyReLU activation, combined with max pooling layers to reduce dimensionality.
- **Latent Space:** Compressed representation of the input image.
- **Decoder:** Convolutional layers with upsampling to reconstruct the original 28×28 image.
- **Loss Function:** Binary cross-entropy, suitable for pixel-wise reconstruction.
- **Optimizer:** Adam optimizer for efficient training.
- **Training Enhancements:** Early stopping and learning rate reduction are used to prevent overfitting and improve convergence.

Case Studies:

1.Handwritten Digit Reconstruction:
Using the MNIST dataset, convolutional autoencoders can effectively reconstruct handwritten digits while preserving their sharpness and structure. Unlike dense autoencoders, which may produce blurred outputs, convolutional models retain the fine details such as edges and curves of the digits. This makes them more reliable for digit recognition tasks.

2.Image Denoising
Convolutional autoencoders can also be applied to denoising problems. By training the model with noisy images as input and clean images as output, the network learns to remove noise and reconstruct cleaner versions of the images. This technique is widely used in image preprocessing and medical imaging applications.

3.Anomaly Detection
The compressed latent representations learned by convolutional autoencoders can be applied to anomaly detection tasks. For example, in fraud detection or defect identification, the model is trained on normal data. When an unusual input is given, the reconstruction error increases significantly, indicating an anomaly.

4.Dimensionality Reduction
Similar to Principal Component Analysis (PCA), convolutional autoencoders can reduce highdimensional image data into compact feature representations. These compressed features can then be used for clustering, visualization, or as input to other machine learning models, thereby improving computational efficiency.

RIYA GOEL 35517711623

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D, BatchNormalization, LeakyReLU, Cropping2D
from tensorflow.keras.models import Model
from tensorflow.keras.datasets import mnist
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau

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

# Normalize and reshape data to (samples, 28, 28, 1)
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.

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

# Build convolutional autoencoder with moderate complexity
input_img = Input(shape=(28, 28, 1))
```

```
# Encoder
x = Conv2D(32, (3, 3), padding='same')(input_img)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = MaxPooling2D((2, 2), padding='same')(x)  # 14x14x32

x = Conv2D(16, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = MaxPooling2D((2, 2), padding='same')(x)  # 7x7x16

x = Conv2D(8, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)  # 4x4x8
```

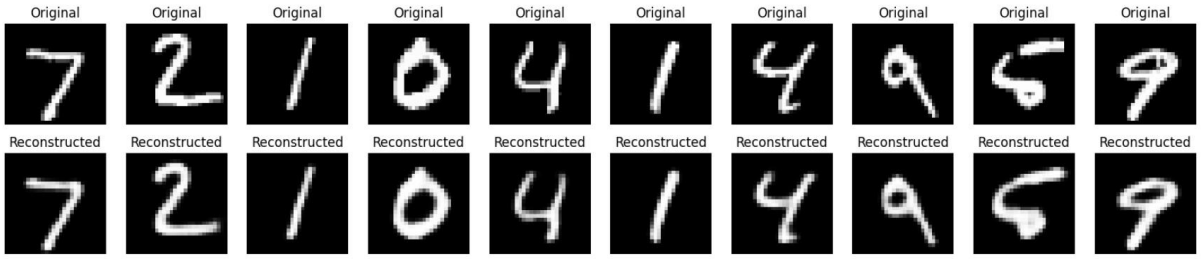
```
# Decoder
x = Conv2D(8, (3, 3), padding='same')(encoded)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = UpSampling2D((2, 2))(x)  # 8x8x8

x = Conv2D(16, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = UpSampling2D((2, 2))(x)  # 16x16x16

x = Conv2D(32, (3, 3), padding='same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = UpSampling2D((2, 2))(x)  # 32x32x32

# Final layer to get 28x28x1
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

# Crop to 28x28
decoded = Cropping2D(((2, 2), (2, 2)))(decoded)

# Model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
```

```
# Callbacks for faster training & avoiding overfitting
early_stop = EarlyStopping(monitor='val_loss', patience=3, restore_best_weights=True)
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=2, min_lr=1e-6)

# Train the autoencoder
autoencoder.fit(
    x_train, x_train,
    epochs=20,
    batch_size=128,
    shuffle=True,
    validation_data=(x_test, x_test),
    callbacks=[early_stop, reduce_lr]
)

# Predict on test set
decoded_imgs = autoencoder.predict(x_test)

# Visualize original and reconstructed images
n = 10  # number of digits to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(28, 28), cmap='gray')
    plt.title("Original")
    plt.axis('off')

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

plt.show()
```

Output:

```
Epoch 1/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 17s 19ms/step - loss: 0.2258 - val_loss: 0.1353 - learning_rate: 0.0010
Epoch 2/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 11s 7ms/step - loss: 0.1163 - val_loss: 0.1048 - learning_rate: 0.0010
Epoch 3/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.1042 - val_loss: 0.1000 - learning_rate: 0.0010
Epoch 4/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 0.0989 - val_loss: 0.0951 - learning_rate: 0.0010
Epoch 5/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 0.0954 - val_loss: 0.0936 - learning_rate: 0.0010
Epoch 6/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.0928 - val_loss: 0.0897 - learning_rate: 0.0010
Epoch 7/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0906 - val_loss: 0.0884 - learning_rate: 0.0010
Epoch 8/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0893 - val_loss: 0.0870 - learning_rate: 0.0010
Epoch 9/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 0.0880 - val_loss: 0.0861 - learning_rate: 0.0010
Epoch 10/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0873 - val_loss: 0.0847 - learning_rate: 0.0010
Epoch 11/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - loss: 0.0863 - val_loss: 0.0889 - learning_rate: 0.0010
Epoch 12/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.0857 - val_loss: 0.0834 - learning_rate: 0.0010
Epoch 13/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0847 - val_loss: 0.0830 - learning_rate: 0.0010
Epoch 14/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - loss: 0.0839 - val_loss: 0.0823 - learning_rate: 0.0010
Epoch 15/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 0.0834 - val_loss: 0.0817 - learning_rate: 0.0010
```

```
Epoch 16/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0830 - val_loss: 0.0819 - learning_rate: 0.0010
Epoch 17/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - loss: 0.0825 - val_loss: 0.0823 - learning_rate: 0.0010
Epoch 18/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0816 - val_loss: 0.0812 - learning_rate: 5.0000e-04
Epoch 19/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 8ms/step - loss: 0.0815 - val_loss: 0.0803 - learning_rate: 5.0000e-04
Epoch 20/20
469/469 ━━━━━━━━━━━━━━━━━━━━ 5s 7ms/step - loss: 0.0812 - val_loss: 0.0800 - learning_rate: 5.0000e-04
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step
```



Learning Outcome:

RIYA GOEL 35517711623

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-dependent, making RNNs an appropriate choice due to their ability to retain information from previous time steps through internal memory. This experiment aims to explore how RNNs can model complex temporal relationships in stock price data and generate accurate predictions of future values. The implementation involves data preprocessing, feature scaling, sequence generation, and training the RNN model on real-world stock datasets. The experiment also seeks to evaluate the model's performance using appropriate error metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE), and to visualize predictions alongside actual stock prices. Ultimately, this experiment provides hands-on experience with time series forecasting using deep learning techniques and highlights the practical use of RNNs in financial market analysis and decision-making.

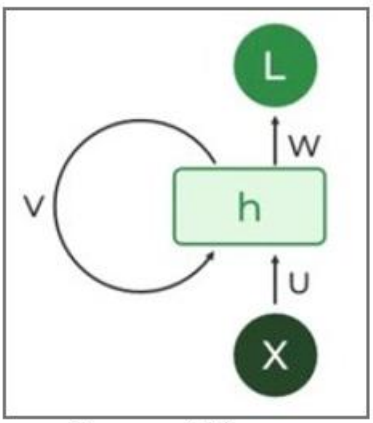
Theory:

Recurrent Neural Networks (RNNs) differ from regular neural networks in how they process information. While standard neural networks pass information in one direction i.e. from input to output, RNNs feed information back into the network at each step.

Key Components of RNNs:

There are mainly two components of RNNs that we will discuss.

1. **Recurrent Neurons:**
   The fundamental processing unit in RNN is a Recurrent Unit. They hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can "remember" information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

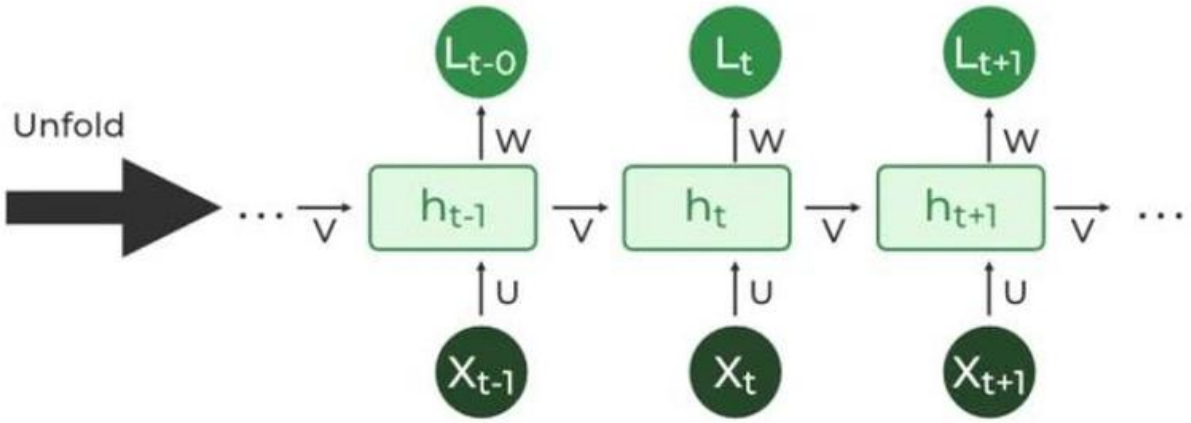


Recurrent Neuron

RNN Unfolding:

RNN unfolding or unrolling is the process of expanding the recurrent structure over time steps. During unfolding each step of the sequence is represented as a separate layer in a series illustrating how information flows across each time step.

This unrolling enables backpropagation through time (BPTT) a learning process where errors are propagated across time steps to adjust the network's weights enhancing the RNN's ability to learn dependencies within sequential data.



RNN Unfolding

Stock Price Prediction Using RNN/LSTM:

In stock price prediction, historical stock prices (e.g., closing prices) are used as sequential inputs to the RNN/LSTM model. The network learns patterns and temporal relationships in the data to predict future prices.

The general workflow is:

- **Data Preprocessing:** Normalize stock prices and prepare sequences of past prices as inputs.
- **Model Design:** Use RNN/LSTM layers to learn temporal dependencies.
- **Training:** Optimize the model by minimizing the prediction error (loss function).
- **Prediction:** Generate future stock price estimates from learned patterns.

Advantages of Recurrent Neural Networks:

- **Sequential Memory:** RNNs retain information from previous inputs making them ideal for time-series predictions where past data is crucial.
- **Enhanced Pixel Neighborhoods:** RNNs can be combined with convolutional layers to capture extended pixel neighborhoods improving performance in image and video data processing.

Limitations of Recurrent Neural Networks (RNNs):

While RNNs excel at handling sequential data they face two main training challenges i.e vanishing gradient and exploding gradient problem:

1. **Vanishing Gradient:** During backpropagation gradients diminish as they pass through each time step leading to minimal weight updates. This limits the RNN's ability to learn long-term dependencies which is crucial for tasks like language translation.
2. **Exploding Gradient:** Sometimes gradients grow uncontrollably causing excessively large weight updates that de-stabilize training.

Applications of Recurrent Neural Networks:
RNNs are used in various applications where data is sequential or time-based:

- **Time-Series Prediction:** RNNs excel in forecasting tasks, such as stock market predictions and weather forecasting.
- **Natural Language Processing (NLP):** RNNs are fundamental in NLP tasks like language modeling, sentiment analysis and machine translation.
- **Speech Recognition:** RNNs capture temporal patterns in speech data, aiding in speechto-text and other audio-related applications.
- **Image and Video Processing:** When combined with convolutional layers, RNNs help analyze video sequences, facial expressions and gesture recognition.

Conclusion:

RNNs, especially LSTM models, offer powerful frameworks for stock price prediction by modeling time series data's sequential nature. With proper data preprocessing and training, they can capture trends and patterns in historical prices to make informed forecasts.
Dataset Used:
The stock price data used in this project was retrieved from Yahoo Finance using the Python yfinance library. Yahoo Finance is a widely-used platform providing historical market data for a wide range of financial instruments including stocks, ETFs, indices, and more.

Source Code:

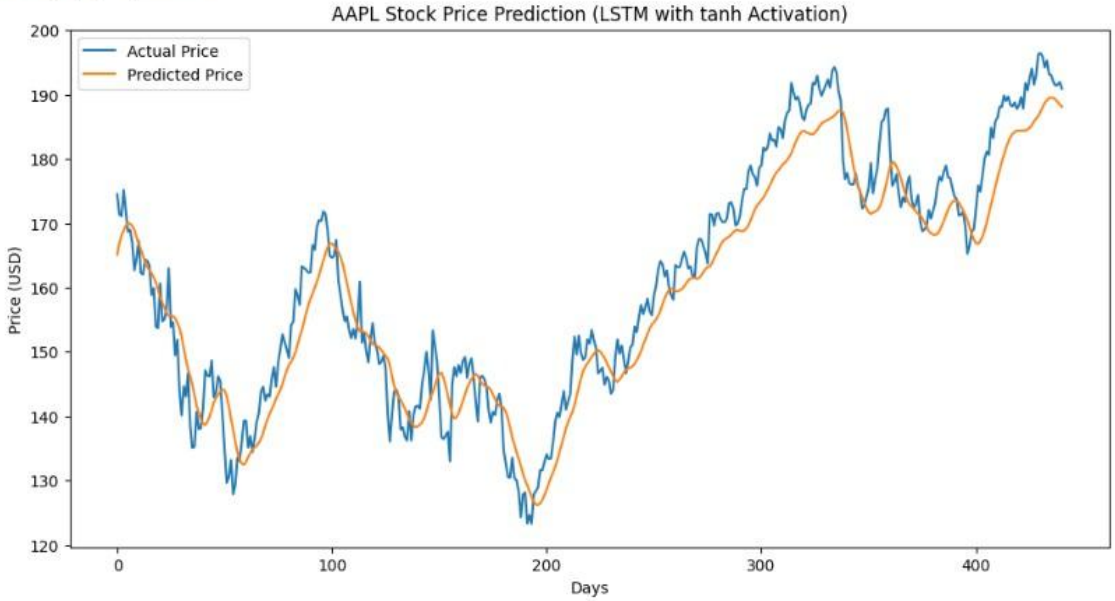
```
import numpy as np
import pandas as pd
import yfinance as yf
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error, mean_absolute_error
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, 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 (LSTM with tanh Activation)')
plt.xlabel('Days')
plt.ylabel('Price (USD)')
plt.legend()
plt.show()
```

Output:

```
/tmp/ipython-input-3462793165.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 completed
/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)
Epoch 1/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 6s 57ms/step - loss: 0.0161 - val_loss: 0.0033
Epoch 2/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 54ms/step - loss: 9.7168e-04 - val_loss: 0.0011
Epoch 3/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 71ms/step - loss: 8.2164e-04 - val_loss: 0.0010
Epoch 4/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 9.6158e-04 - val_loss: 0.0021
Epoch 5/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 52ms/step - loss: 7.6708e-04 - val_loss: 0.0038
Epoch 6/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 69ms/step - loss: 8.9633e-04 - val_loss: 0.0011
Epoch 7/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 50ms/step - loss: 8.3866e-04 - val_loss: 0.0012
Epoch 8/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 7.5211e-04 - val_loss: 8.8208e-04
Epoch 9/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 6s 69ms/step - loss: 8.3622e-04 - val_loss: 0.0049
Epoch 10/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 54ms/step - loss: 8.2805e-04 - val_loss: 0.0014
Epoch 11/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 5s 51ms/step - loss: 5.8778e-04 - val_loss: 0.0020
Epoch 12/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 7.6371e-04 - val_loss: 9.0270e-04
Epoch 13/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 76ms/step - loss: 5.6713e-04 - val_loss: 0.0020
Epoch 14/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 5.1804e-04 - val_loss: 0.0023
Epoch 15/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 6.7295e-04 - val_loss: 0.0015
Epoch 16/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 6s 60ms/step - loss: 6.8633e-04 - val_loss: 0.0012
Epoch 17/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 64ms/step - loss: 7.0896e-04 - val_loss: 7.5957e-04
Epoch 18/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 4s 50ms/step - loss: 5.5513e-04 - val_loss: 0.0010
Epoch 19/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 3s 51ms/step - loss: 5.3561e-04 - val_loss: 7.3265e-04
Epoch 20/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 6s 59ms/step - loss: 5.4722e-04 - val_loss: 8.4190e-04
14/14 ━━━━━━━━━━━━━━━━━━━━ 1s 35ms/step
MSE: 29.8657
MAE: 4.6519
Accuracy (approx): 97.10%
```

```
MSE: 29.8657
MAE: 4.6519
Accuracy (approx): 97.10%
```



Learning Outcome:

RIYA GOEL 35517711623

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.

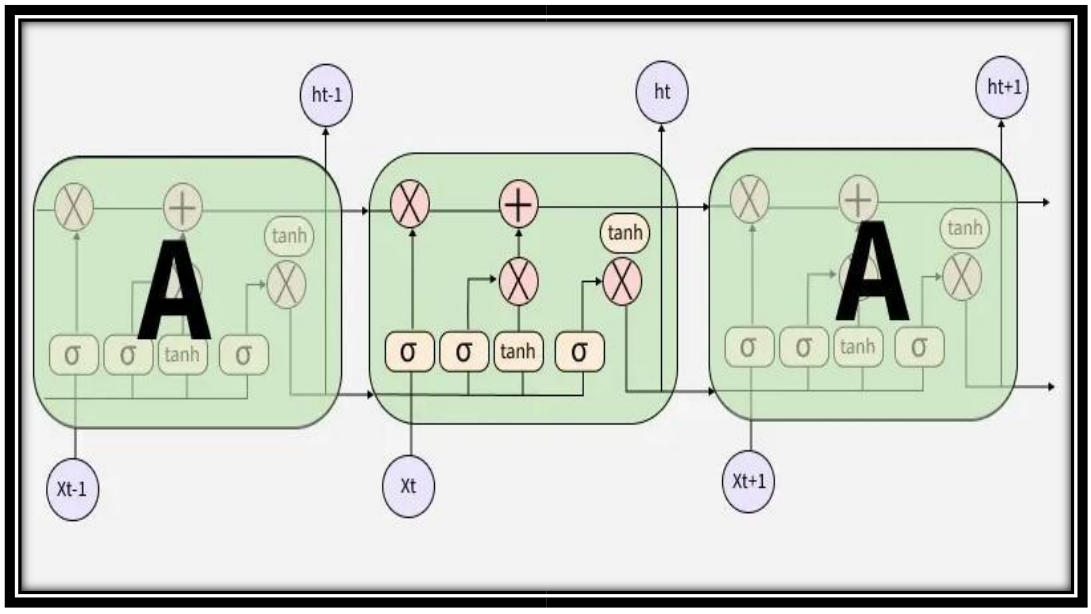
Theory:

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
```

RIYA GOEL 35517711623

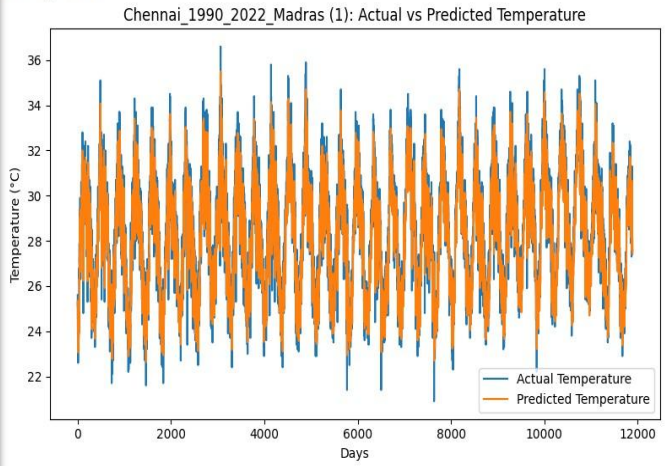
```python
def plot_predictions(actual, predicted, city_name):
    plt.figure(figsize=(8, 5))
    plt.plot(actual, label='Actual Temperature')
    plt.plot(predicted, label='Predicted Temperature')
    plt.xlabel('Days')
    plt.ylabel('Temperature (°C)')
    plt.title(f'{city_name}: Actual vs Predicted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

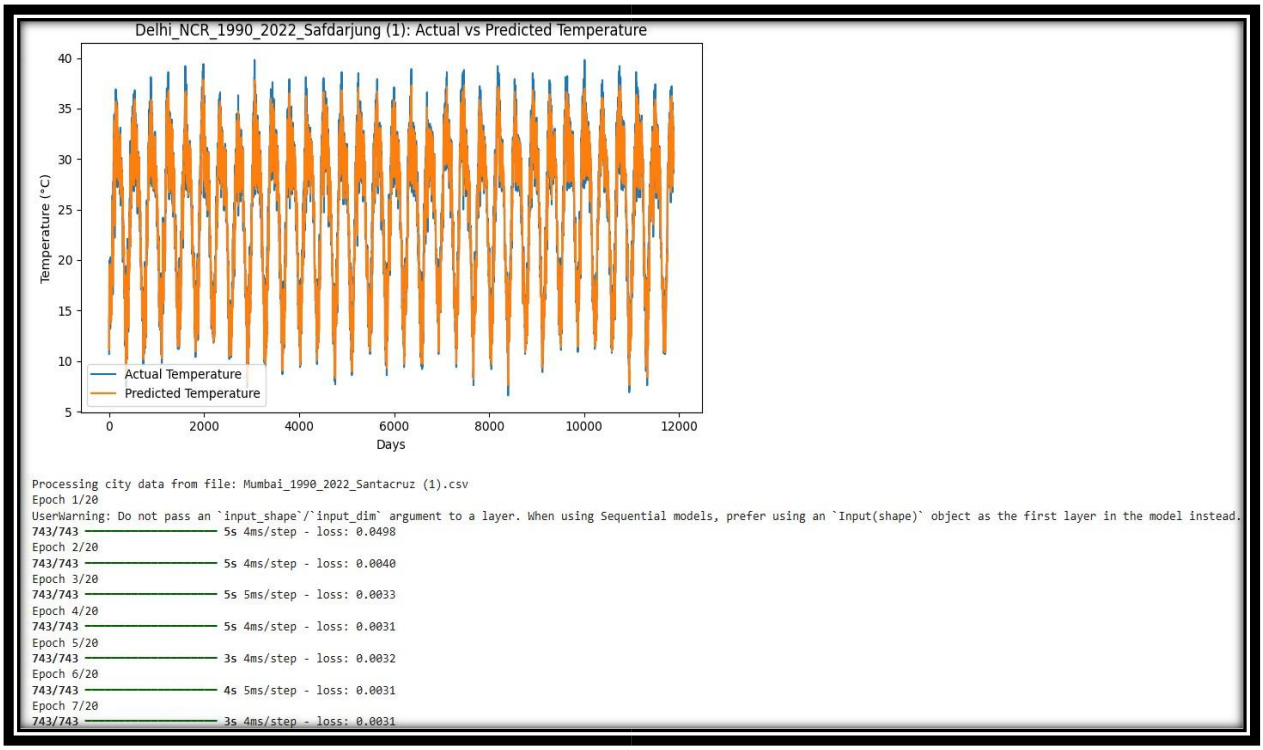
RIYA GOEL 35517711623

```
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%
```



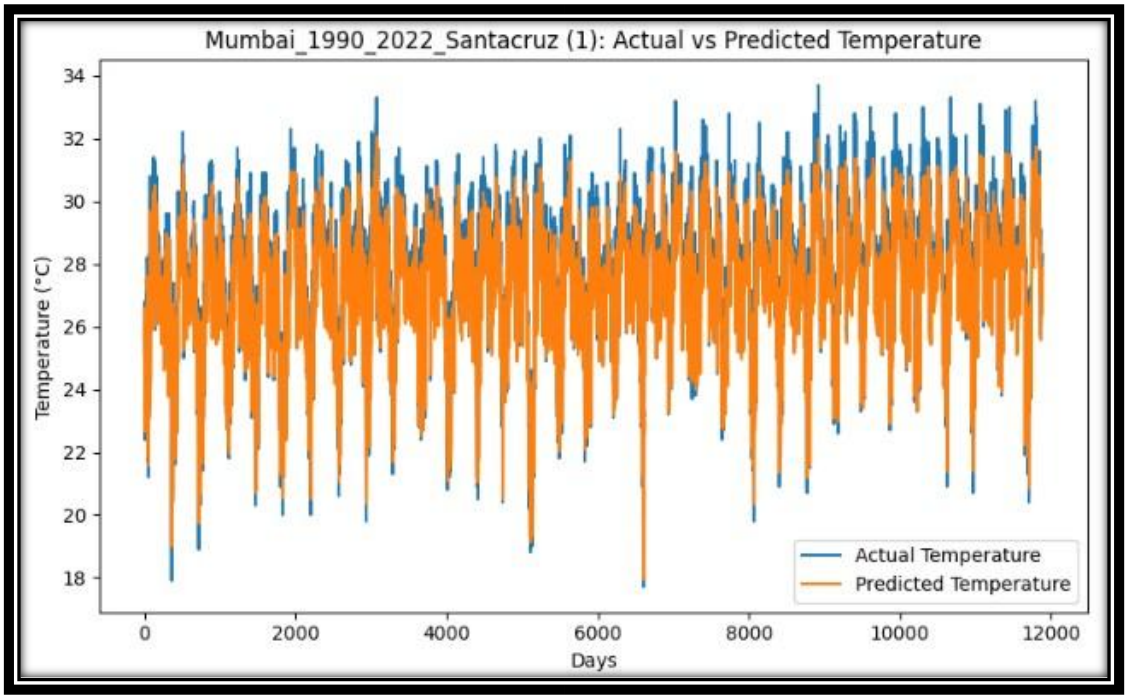

```
Processing city data from file: Delhi_NCR_1990_2022_Safdarjung (1).csv
Epoch 1/20
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
743/743 ━━━━━━━━━━━━━━━━━━━━ 6s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

RIYA GOEL 35517711623




```
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:

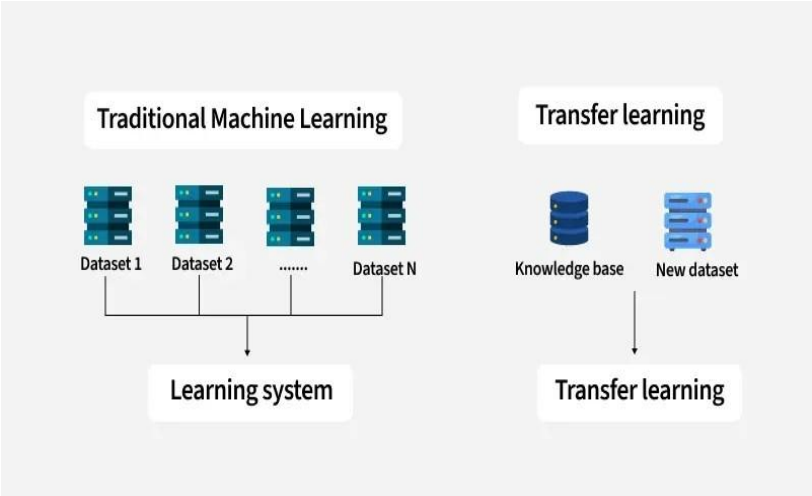
EXPERIMENT-9

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

Theory:

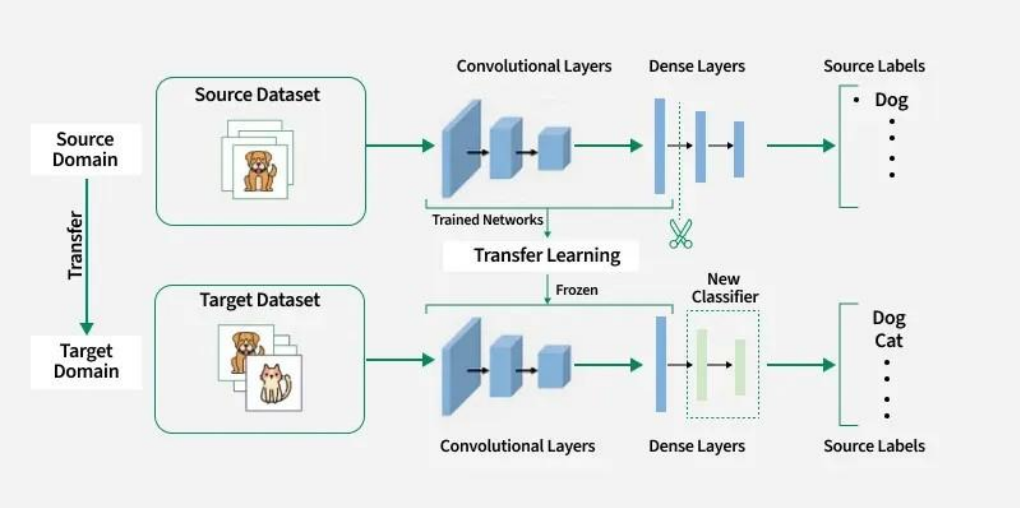
1. Introduction

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



2. Importance of Transfer Learning

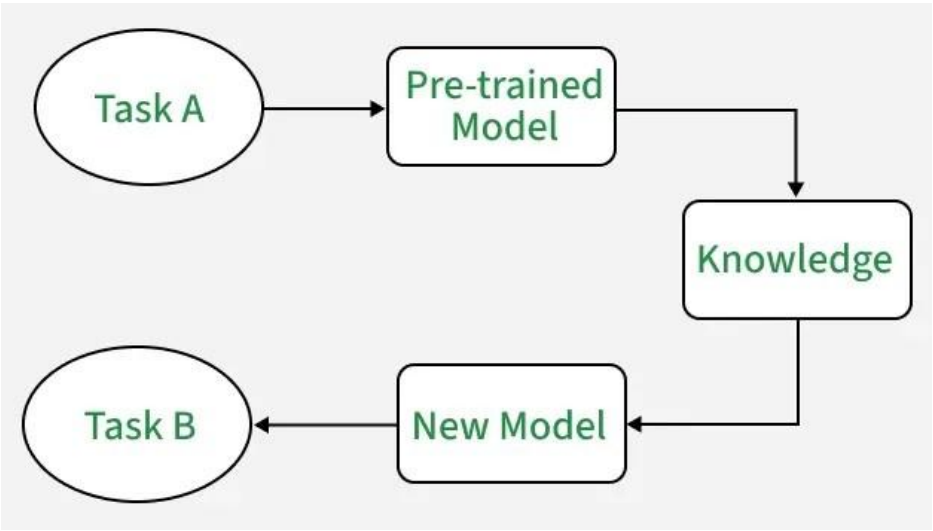
- **Limited Data Handling:** In many real-world problems, collecting large labeled datasets is challenging. Transfer Learning allows the use of pre-trained models that have already learned generalized patterns, significantly reducing the need for extensive data.
- **Improved Accuracy:** Since pre-trained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



3. Working of Transfer Learning

The process of Transfer Learning typically involves the following stages:

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



Code:

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

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

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

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

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

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

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

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

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

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

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

RIYA GOEL 35517711623

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

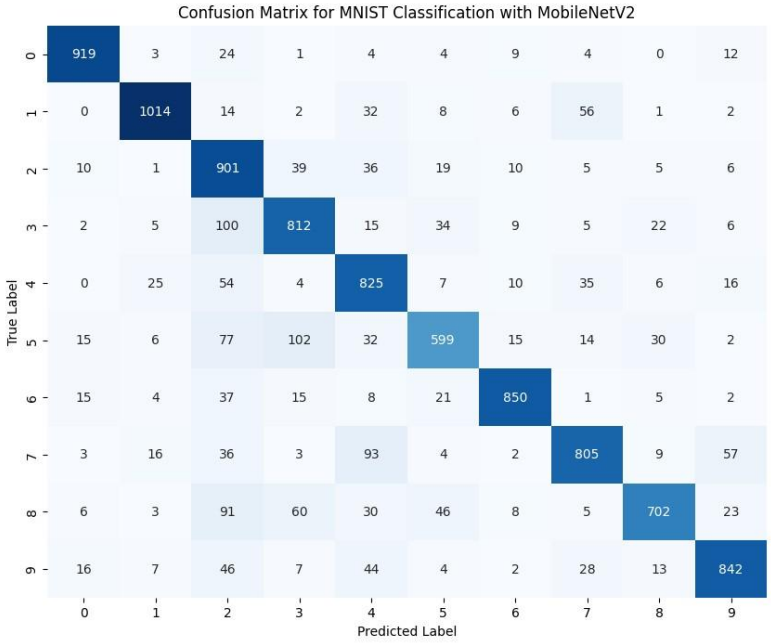
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.8117 - loss: 0.6716
Test loss: 0.6333
Test accuracy: 0.8269
```

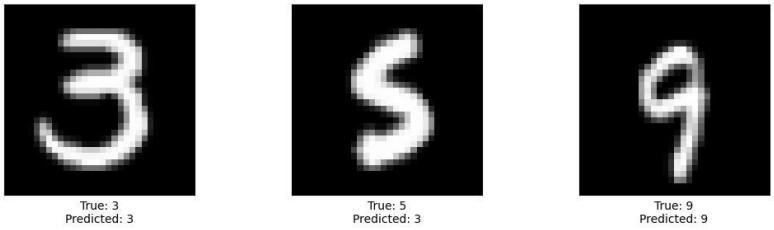
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

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

# Confusion Matrix
cm = confusion_matrix(test_true_classes, test_predictions_classes)

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

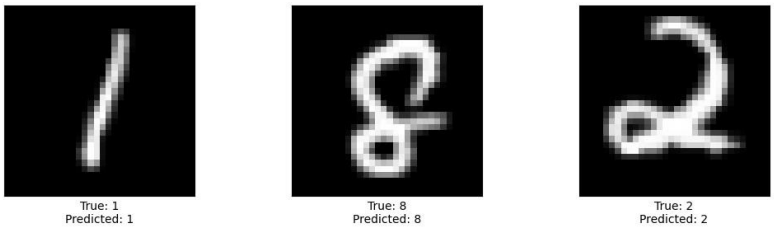
```
# Step 7: Visualize Sample Predictions
def display_sample(sample_images, sample_labels, sample_predictions):
    fig, axes = plt.subplots(3, 3, figsize=(12, 12))
    fig.subplots_adjust(hspace=0.5, wspace=0.5)

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

    plt.show()

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

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

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

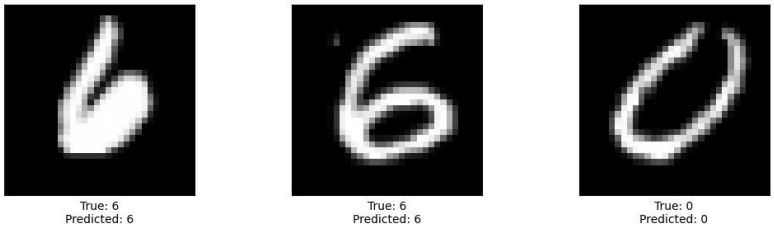
RIYA GOEL 35517711623

Output:









Learning Outcome:

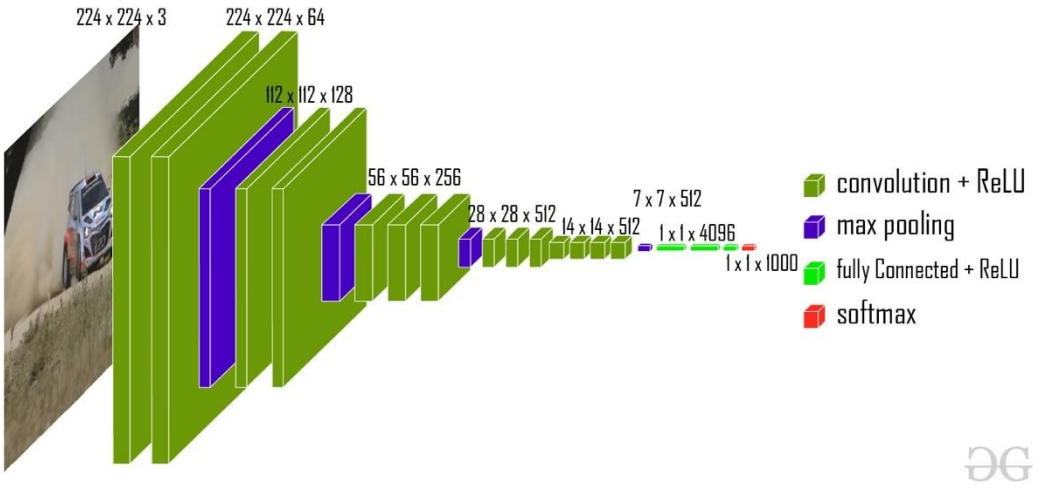
EXPERIMENT-10

Aim: Implementation of transfer learning using pre-trained model (VCG16) for image classification in python.

Theory:

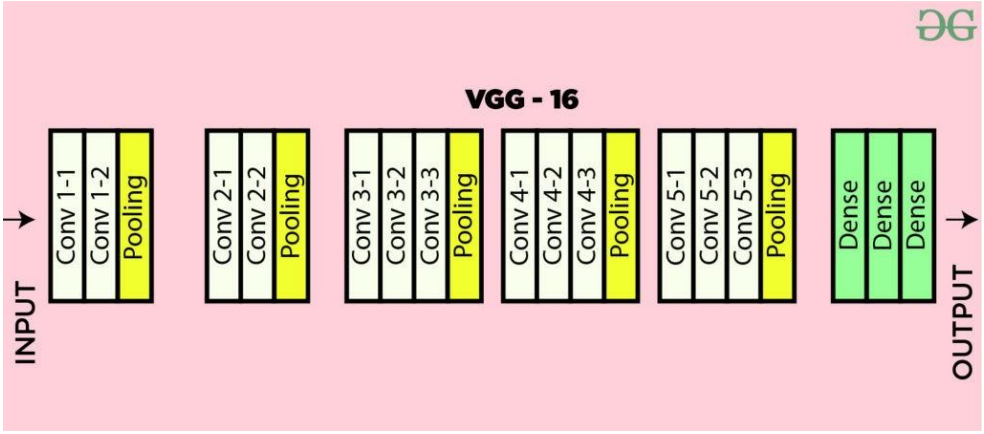
Introduction

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



VGG Architecture

The VGG-16 architecture is a deep convolutional neural network (CNN) designed for image classification tasks. VGG-16 is characterized by its simplicity and uniform architecture, making it easy to understand and implement.
It typically consists of 16 layers, including 13 convolutional layers and 3 fully connected layers. These layers are organized into blocks, with each block containing multiple convolutional layers followed by a max-pooling layer for downsampling.



Here's a breakdown of the VGG-16 architecture based on the provided details:

1. **Input Layer:**
- Input dimensions: (224, 224, 3)
2. **Convolutional Layers (64 filters, 3x3 filters, same padding):**
- Two consecutive convolutional layers with 64 filters each and a filter size of 3x3.
- Same padding is applied to maintain spatial dimensions.
3. **Max Pooling Layer (2x2, stride 2):**
- Max-pooling layer with a pool size of 2x2 and a stride of 2.
4. **Convolutional Layers (128 filters, 3x3 filters, same padding):**
- Two consecutive convolutional layers with 128 filters each and a filter size of 3x3.
5. **Max Pooling Layer (2x2, stride 2):**
- Max-pooling layer with a pool size of 2x2 and a stride of 2.
6. **Convolutional Layers (256 filters, 3x3 filters, same padding):**
- Two consecutive convolutional layers with 256 filters each and a filter size of 3x3.
7. **Convolutional Layers (512 filters, 3x3 filters, same padding):**
- Two sets of three consecutive convolutional layers with 512 filters each and a filter size of 3x3.
8. **Max Pooling Layer (2x2, stride 2):**
- Max-pooling layer with a pool size of 2x2 and a stride of 2.
9. **Stack of Convolutional Layers and Max Pooling:**
- Two additional convolutional layers after the previous stack.
- Filter size: 3x3.
10. **Flattening:**
- Flatten the output feature map (7x7x512) into a vector of size 25088.
11. **Fully Connected Layers:**
- Three fully connected layers with ReLU activation.
- First layer with input size 25088 and output size 4096.
- Second layer with input size 4096 and output size 4096.
- Third layer with input size 4096 and output size 1000, corresponding to the 1000 classes in the ILSVRC challenge.
- Softmax activation is applied to the output of the third fully connected layer for classification.

This architecture follows the specifications provided, including the use of ReLU activation function and the final fully connected layer outputting probabilities for 1000 classes using softmax activation.

Algorithm: Transfer Learning using VGG16 Model

Input:

- Input Image: A 224×224 RGB image (3 channels).
- Pre-trained Model: VGG16 architecture with ImageNet weights.
-

Architecture:

1.Convolutional Blocks (Block 1 to Block 5):

- Each block consists of multiple convolutional layers with 3×3 filters.
- The number of filters increases with network depth (e.g., 64, 128, 256, 512).
- ReLU activation is applied after each convolution to introduce non-linearity.
- Batch Normalization may be used for improved training stability and faster convergence.

2.Max-Pooling Layers:

- After each convolutional block, a max-pooling layer with a 2×2 filter and stride of 2 is applied.
- This operation reduces spatial dimensions while retaining the most prominent features.

3.Fully Connected Layers:

- The convolutional base is followed by three dense layers.
- The first two fully connected layers have 4096 units each with ReLU activation. • The final (output) layer contains units equal to the number of classes and uses Softmax activation to generate probabilities.

4.Transfer Learning Integration:

- Load the pre-trained VGG16 model trained on the ImageNet dataset.
- Remove the original top classification layers (include_top=False).
- Add custom dense layers for the target dataset.
- Freeze lower convolutional layers to retain general feature extraction.
- Fine-tune higher layers using the new dataset for task-specific adaptation.

5.Training:

- Use a categorical cross-entropy loss function suitable for multi-class classification.
- Optimize using SGD or Adam optimizer.
- Update weights through backpropagation to minimize classification error.
- Monitor performance using accuracy and validation loss metrics.

6.Prediction:

- Input a new image (resized to 224×224×3).
- Forward propagate the image through the network.
- The model outputs class probabilities via the Softmax layer.
- The class with the highest probability is selected as the final prediction.

Challenges:

- VGG16 has a large number of parameters, leading to high computational cost.
- Requires significant memory and processing power, limiting deployment on low-resource devices.

Code:

```
# ============================================================
# 🌿 Cotton Disease Classification using MobileNetV2 (Improved)
# ============================================================

# ------------------------------
# 1️⃣ Unzip the dataset
# ------------------------------
import zipfile
import os

zip_path = 'cotton_disease.zip'
extract_path = '.'
dataset_folder = os.path.join(extract_path, 'cotton_disease')

if not os.path.exists(dataset_folder):
    if os.path.exists(zip_path):
        with zipfile.ZipFile(zip_path, 'r') as zip_ref:
            zip_ref.extractall(extract_path)
        print("✅ Dataset extracted successfully at:", dataset_folder)
    else:
        raise FileNotFoundError(f"{zip_path} not found. Place it in the current directory.")
else:
    print("✅ Dataset already exists at:", dataset_folder)
```

```
# ------------------------------
# 2️⃣ Import libraries
# ------------------------------
import tensorflow as tf
print("TensorFlow Version:", tf.__version__)

from tensorflow.compat.v1 import ConfigProto, InteractiveSession
config = ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = 0.5
config.gpu_options.allow_growth = True
session = InteractiveSession(config=config)

from tensorflow.keras.layers import Dense, GlobalAveragePooling2D, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.applications import MobileNetV2
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import matplotlib.pyplot as plt
from glob import glob

# ------------------------------
# 3️⃣ Set parameters
# ------------------------------
IMAGE_SIZE = [128, 128]
train_path = 'cotton_disease/train'
valid_path = 'cotton_disease/test'

# ------------------------------
# 4️⃣ Load pre-trained MobileNetV2
# ------------------------------
base_model = MobileNetV2(input_shape=IMAGE_SIZE + [3], weights='imagenet', include_top=False)
base_model.trainable = False  # Freeze base layers

folders = glob(train_path + '/*')
num_classes = len(folders)
print("Number of Classes:", num_classes)
```

RIYA GOEL 35517711623

```
# ------------------------------
# 5️⃣ Build the model
# ------------------------------
x = base_model.output
x = GlobalAveragePooling2D()(x)
x = Dense(512, activation='relu')(x)
x = Dropout(0.3)(x)  # Helps reduce overfitting
prediction = Dense(num_classes, activation='softmax')(x)

model = Model(inputs=base_model.input, outputs=prediction)
model.summary()

# ------------------------------
# 6️⃣ Compile the model (Lower LR)
# ------------------------------
model.compile(
    loss='categorical_crossentropy',
    optimizer=Adam(learning_rate=1e-4),
    metrics=['accuracy']
)

# ------------------------------
# 7️⃣ Data Preprocessing & Augmentation
# ------------------------------
train_datagen = ImageDataGenerator(
    rescale=1./255,
    shear_range=0.3,
    zoom_range=0.3,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True
)

test_datagen = ImageDataGenerator(rescale=1./255)

training_set = train_datagen.flow_from_directory(
    train_path,
    target_size=IMAGE_SIZE,
    batch_size=32,
    class_mode='categorical'
)


        test_set = test_datagen.flow_from_directory(
            valid_path,
            target_size=IMAGE_SIZE,
            batch_size=32,
            class_mode='categorical'
        )

        # ------------------------------
        # 8️⃣ Callbacks
        # ------------------------------
        early_stop = EarlyStopping(
            monitor='val_loss',
            patience=5,
            restore_best_weights=True
        )
        reduce_lr = ReduceLROnPlateau(
            monitor='val_loss',
            factor=0.5,
            patience=3,
            min_lr=1e-6
        )

        # ------------------------------
        # 9️⃣ Train the model
        # ------------------------------
        r = model.fit(
            training_set,
            validation_data=test_set,
            epochs=50,
            steps_per_epoch=len(training_set),
            validation_steps=len(test_set),
            callbacks=[early_stop, reduce_lr]
        )
```

✅ Dataset already exists at: ./cotton_disease
TensorFlow Version: 2.20.0
Number of Classes: 4
Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_layer_1 (InputLayer)	(None, 128, 128, 3)	0	-
Conv1 (Conv2D)	(None, 64, 64, 32)	864	input_layer_1[0]…
bn_Conv1 (BatchNormalizatio…	(None, 64, 64, 32)	128	Conv1[0][0]
Conv1_relu (ReLU)	(None, 64, 64, 32)	0	bn_Conv1[0][0]
expanded_conv_dept… (DepthwiseConv2D)	(None, 64, 64, 32)	288	Conv1_relu[0][0]
expanded_conv_dept… (BatchNormalizatio…	(None, 64, 64, 32)	128	expanded_conv_de…
expanded_conv_dept… (ReLU)	(None, 64, 64, 32)	0	expanded_conv_de…
expanded_conv_proj… (Conv2D)	(None, 64, 64, 16)	512	expanded_conv_de…
expanded_conv_proj… (BatchNormalizatio…	(None, 64, 64, 16)	64	expanded_conv_pr…
block_16_depthwise… (ReLU)	(None, 4, 4, 960)	0	block_16_depthwi…
block_16_project (Conv2D)	(None, 4, 4, 320)	307,200	block_16_depthwi…
block_16_project_BN (BatchNormalizatio…	(None, 4, 4, 320)	1,280	block_16_project…
Conv_1 (Conv2D)	(None, 4, 4, 1280)	409,600	block_16_project…
Conv_1_bn (BatchNormalizatio…	(None, 4, 4, 1280)	5,120	Conv_1[0][0]
out_relu (ReLU)	(None, 4, 4, 1280)	0	Conv_1_bn[0][0]
global_average_poo… (GlobalAveragePool…	(None, 1280)	0	out_relu[0][0]
dense_2 (Dense)	(None, 512)	655,872	global_average_p…
dropout (Dropout)	(None, 512)	0	dense_2[0][0]
dense_3 (Dense)	(None, 4)	2,052	dropout[0][0]

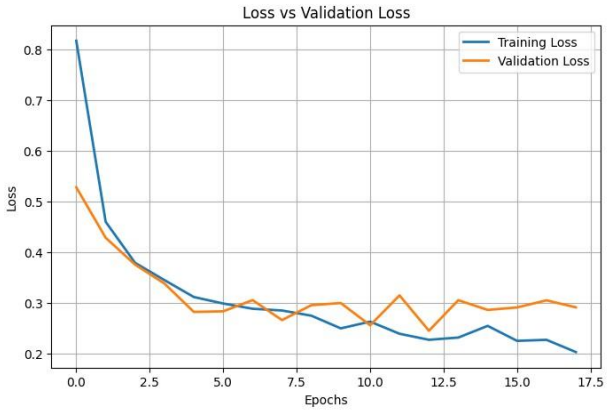
Total params: 2,915,908 (11.12 MB)
Trainable params: 657,924 (2.51 MB)
Non-trainable params: 2,257,984 (8.61 MB)
Found 1951 images belonging to 4 classes.
Found 106 images belonging to 4 classes.

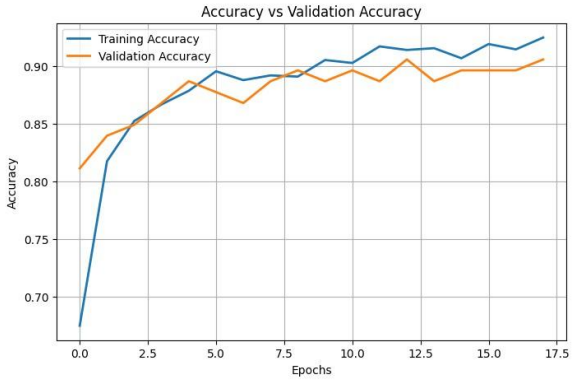
```
Epoch 11/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 249ms/step - accuracy: 0.9024 - loss: 0.2615 - val_accuracy: 0.8962 - val_loss: 0.2564 - learning_rate: 1.0000e-04
Epoch 12/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 242ms/step - accuracy: 0.9213 - loss: 0.2398 - val_accuracy: 0.8868 - val_loss: 0.3152 - learning_rate: 1.0000e-04
Epoch 13/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 249ms/step - accuracy: 0.9111 - loss: 0.2463 - val_accuracy: 0.9057 - val_loss: 0.2456 - learning_rate: 1.0000e-04
Epoch 14/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 243ms/step - accuracy: 0.9214 - loss: 0.2210 - val_accuracy: 0.8868 - val_loss: 0.3058 - learning_rate: 1.0000e-04
Epoch 15/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 251ms/step - accuracy: 0.9136 - loss: 0.2443 - val_accuracy: 0.8962 - val_loss: 0.2867 - learning_rate: 1.0000e-04
Epoch 16/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 239ms/step - accuracy: 0.9173 - loss: 0.2246 - val_accuracy: 0.8962 - val_loss: 0.2915 - learning_rate: 1.0000e-04
Epoch 17/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 246ms/step - accuracy: 0.9005 - loss: 0.2448 - val_accuracy: 0.8962 - val_loss: 0.3055 - learning_rate: 5.0000e-05
Epoch 18/50
61/61 ━━━━━━━━━━━━━━━━━━━━ 15s 244ms/step - accuracy: 0.9224 - loss: 0.2004 - val_accuracy: 0.9057 - val_loss: 0.2917 - learning_rate: 5.0000e-05
/usr/local/lib/python3.12/dist-packages/IPython/core/pylabtools.py:151: UserWarning: Glyph 128201 (\N{CHART WITH DOWNWARDS TREND}) missing from font(s) DejaVu Sans.
  fig.canvas.print_figure(bytes_io, **kw)
```

RIYA GOEL 35517711623

```
import matplotlib.pyplot as plt

# Plot Loss
plt.figure(figsize=(8,5))
plt.plot(r.history['loss'], label='Training Loss', linewidth=2)
plt.plot(r.history['val_loss'], label='Validation Loss', linewidth=2)
plt.legend()
plt.title("Loss vs Validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.grid(True)
plt.show()
plt.savefig('Loss_ValLoss_Unsmoothed.png')

# Plot Accuracy
plt.figure(figsize=(8,5))
plt.plot(r.history['accuracy'], label='Training Accuracy', linewidth=2)
plt.plot(r.history['val_accuracy'], label='Validation Accuracy', linewidth=2)
plt.legend()
plt.title("Accuracy vs Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
plt.savefig('Acc_ValAcc_Unsmoothed.png')
```

Output:





Learning Outcome: