**Experiment 1**

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

**Objectives:**

* Understand the foundational functionalities of TensorFlow and Keras for deep learning.
* Explore how to build, compile, and train basic neural network models using Keras.

**Theory:**

TensorFlow and Keras are powerful libraries in Python that form the backbone of deep learning model implementation. They provide a comprehensive ecosystem for building, training, and deploying machine learning models. These libraries are pivotal for both academic and practical experiments in the field of deep learning.

**1. TensorFlow:**

TensorFlow, developed by Google, is an open-source machine learning framework designed for high-performance numerical computations. It enables researchers and developers to efficiently create and train machine learning models across various platforms. TensorFlow's core strength lies in its ability to handle tensors, which are multi-dimensional arrays that form the foundation of all computations in deep learning.

Key features of TensorFlow include:

* **Automatic Differentiation**: TensorFlow computes gradients automatically, simplifying the process of backpropagation for complex neural networks.
* **GPU/TPU Acceleration**: TensorFlow optimally utilizes hardware resources like GPUs and TPUs for faster training and inference.
* **Scalability**: Its distributed training capabilities allow it to handle large-scale models and datasets across multiple devices and servers.
* **Versatility**: TensorFlow supports both low-level control for custom operations and high-level APIs like Keras for simplicity.

**2. Keras:**

Keras is a high-level neural network API integrated with TensorFlow. It is designed to be user-friendly while still providing powerful features for advanced customization. Keras reduces the complexity of building deep learning models by offering intuitive syntax and ready-to-use components for common tasks.

Some of the features of Keras include:

* **Simplified Model Building**: Keras offers Sequential and Functional APIs to define models effortlessly.
* **Ease of Use**: It includes pre-built layers, loss functions, optimizers, and metrics, allowing for rapid prototyping.
* **Modularity**: Every component in Keras is independent, making it easy to adapt and customize.
* **Integration**: Since it is built on TensorFlow, it inherits the performance and scalability benefits of TensorFlow.

**3. Core Concepts:**

* **Tensors**:  
  The primary data structure in TensorFlow, tensors represent multi-dimensional arrays that enable efficient numerical computation. Operations on tensors are highly optimized for both CPU and GPU execution.
* **Layers**:  
  Layers are the building blocks of a deep learning model. In Keras, layers perform specific transformations, such as linear computations, activation functions, or normalization, to process input data progressively.
* **Compilation and Training**:

Keras provides a streamlined workflow for compiling models, training them on datasets, and evaluating their performance. Functions like compile(), fit(), and evaluate() abstract the complexities of training, making the process user-friendly.

* **Backpropagation and Optimization**:

TensorFlow and Keras automate the process of calculating gradients and updating model parameters using optimizers like SGD, Adam, or RMSprop. This minimizes loss functions and improves model accuracy over successive iterations.

**4. Regression and Classification with TensorFlow and Keras**

* **Regression**:  
  In regression tasks, the model predicts continuous outputs. TensorFlow and Keras simplify the creation of regression models by providing built-in loss functions (e.g., Mean Squared Error) and metrics to evaluate model performance.
* **Classification**:  
  For classification tasks, the model predicts categorical outputs, such as labels for data points. TensorFlow and Keras support multi-class and binary classification through layers like Dense, activation functions like softmax or sigmoid, and loss functions like categorical\_crossentropy.

By exploring TensorFlow and Keras, learners gain hands-on experience with key deep learning principles. This experiment lays the groundwork for understanding tensor operations, building neural networks, and solving real-world problems through regression and classification tasks. These tools bridge the gap between theoretical concepts and practical applications in the realm of deep learning.

**Code:**

import numpy as np

import pandas as pd

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential, Model

from tensorflow.keras.layers import Dense, Input

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

# Load Life Expectancy Dataset (Regression)

life\_data = pd.read\_csv("Life Expectancy Data.csv")

life\_data.columns = life\_data.columns.str.strip()  # Remove spaces from column names

life\_data.dropna(inplace=True)

# Feature selection and preprocessing

X\_life = life\_data.drop(columns=["Life expectancy"])

y\_life = life\_data["Life expectancy"]

X\_life = pd.get\_dummies(X\_life)  # Convert categorical to numerical

scaler = StandardScaler()

X\_life = scaler.fit\_transform(X\_life)

# Split data

X\_train\_life, X\_test\_life, y\_train\_life, y\_test\_life = train\_test\_split(X\_life, y\_life, test\_size=0.2, random\_state=42)

# Sequential Model for Regression

seq\_model\_life = Sequential([

    Dense(64, activation='relu', input\_shape=(X\_train\_life.shape[1],)),

    Dense(32, activation='relu'),

    Dense(1)

])

seq\_model\_life.compile(optimizer='SGD', loss='mse')

seq\_model\_life.fit(X\_train\_life, y\_train\_life, epochs=50, batch\_size=32, verbose=1)

# Evaluate Regression Model

test\_mse\_life = seq\_model\_life.evaluate(X\_test\_life, y\_test\_life, verbose=1)

print(f"Test MSE (Sequential Regression): {test\_mse\_life}")

# Functional Model for Regression

inputs = Input(shape=(X\_train\_life.shape[1],))

x = Dense(64, activation='relu')(inputs)

x = Dense(32, activation='relu')(x)

outputs = Dense(1)(x)

func\_model\_life = Model(inputs, outputs)

func\_model\_life.compile(optimizer='SGD', loss='mse')

func\_model\_life.fit(X\_train\_life, y\_train\_life, epochs=50, batch\_size=32, verbose=1)

# Evaluate Functional Regression Model

test\_mse\_func\_life = func\_model\_life.evaluate(X\_test\_life, y\_test\_life, verbose=1)

print(f"Test MSE (Functional Regression): {test\_mse\_func\_life}")

# Load Titanic Dataset (Classification)

titanic\_data = pd.read\_csv("train.csv")

titanic\_data.dropna(subset=["Age", "Embarked"], inplace=True)

# Feature selection and preprocessing

X\_titanic = titanic\_data[["Pclass", "Age", "SibSp", "Parch", "Fare"]]

y\_titanic = titanic\_data["Survived"]

scaler = StandardScaler()

X\_titanic = scaler.fit\_transform(X\_titanic)

# Split data

X\_train\_titanic, X\_test\_titanic, y\_train\_titanic, y\_test\_titanic = train\_test\_split(X\_titanic, y\_titanic, test\_size=0.2, random\_state=42)

# Sequential Model for Classification

seq\_model\_titanic = Sequential([

    Dense(64, activation='relu', input\_shape=(X\_train\_titanic.shape[1],)),

    Dense(32, activation='relu'),

    Dense(1, activation='sigmoid')

])

seq\_model\_titanic.compile(optimizer='SGD', loss='mse', metrics=['accuracy'])

seq\_model\_titanic.fit(X\_train\_titanic, y\_train\_titanic, epochs=50, batch\_size=32, verbose=1)

# Evaluate Classification Model

test\_loss\_seq, test\_acc\_seq = seq\_model\_titanic.evaluate(X\_test\_titanic, y\_test\_titanic, verbose=1)

print(f"Test Loss (Sequential Classification): {test\_loss\_seq}, Test Accuracy: {test\_acc\_seq}")

# Functional Model for Classification

inputs = Input(shape=(X\_train\_titanic.shape[1],))

x = Dense(64, activation='relu')(inputs)

x = Dense(32, activation='relu')(x)

outputs = Dense(1, activation='sigmoid')(x)

func\_model\_titanic = Model(inputs, outputs)

func\_model\_titanic.compile(optimizer='SGD', loss='mse', metrics=['accuracy'])

func\_model\_titanic.fit(X\_train\_titanic, y\_train\_titanic, epochs=50, batch\_size=32, verbose=1)

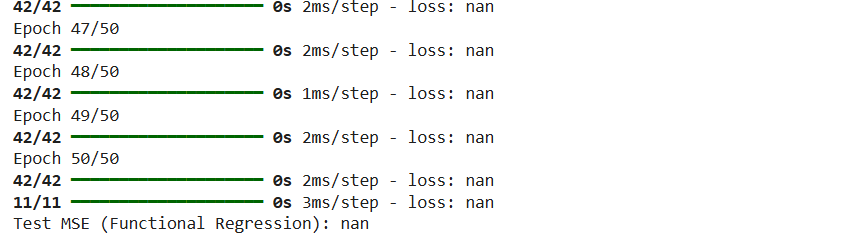
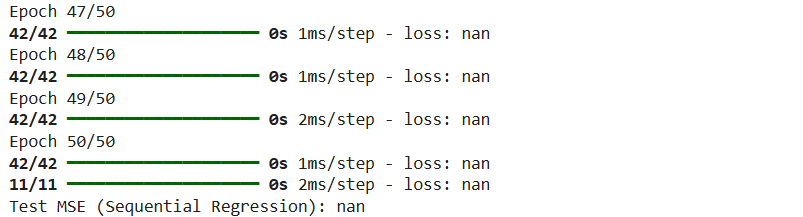
# Evaluate Functional Classification Model

test\_loss\_func, test\_acc\_func = func\_model\_titanic.evaluate(X\_test\_titanic, y\_test\_titanic, verbose=1)

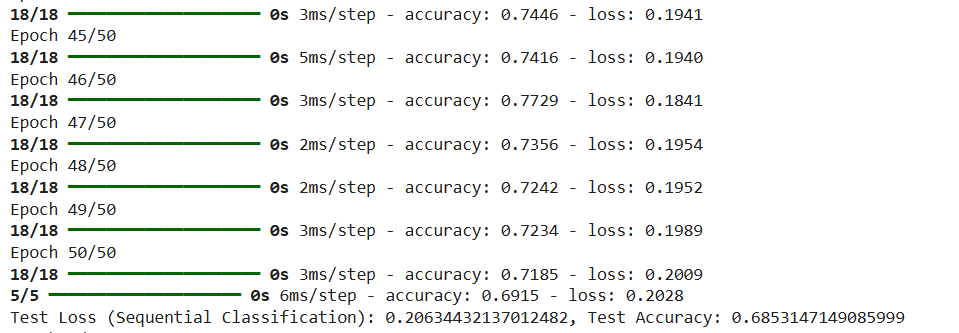
print(f"Test Loss (Functional Classification): {test\_loss\_func}, Test Accuracy: {test\_acc\_func}")

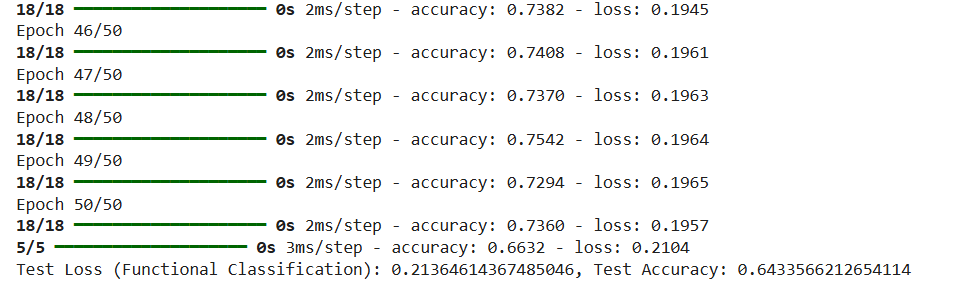
**Output:**

**# Regression**

****

**# Classification**





**Learning Outcomes:**

1. Understand how to build and train regression and classification models using TensorFlow and Keras.
2. Gain insights into key evaluation metrics like Mean Squared Error (MSE) for regression and Accuracy or F1-Score for classification tasks.
3. Develop the ability to preprocess data and apply activation functions, optimizers, and loss functions suitable for different problem domains.

**Experiment 2**

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

**Objectives:**

* To implement an ANN model in Python for regression and evaluate its performance using MSE.
* To implement an ANN model in Python for classification and assess its accuracy using performance metrics.

**Theory:**

**Introduction to Artificial Neural Networks (ANNs)**

Artificial Neural Networks (ANNs) are computational models inspired by the human brain's structure and functioning. They consist of layers of interconnected nodes (neurons) which process and transform input data through various layers to produce an output. The power of ANNs lies in their ability to learn from data, identify patterns, and generalize predictions, making them highly suitable for various tasks such as regression, classification, and time series forecasting.

ANNs are composed of three primary types of layers:

1. **Input Layer**: Receives the raw input data.
2. **Hidden Layers**: Contain neurons that process the data using weights and activation functions.
3. **Output Layer**: Provides the final result or prediction.

Each neuron in the hidden layers and output layer processes inputs from previous layers and passes them through an activation function, which introduces non-linearity into the model. The weights of the connections between neurons are learned during training by using optimization techniques like gradient descent.

**Regression Problems with ANN**

In regression tasks, the objective is to predict a continuous numerical output based on input features. For example, predicting house prices based on location, size, and other features is a typical regression problem.

The basic architecture of an ANN for regression consists of:

* **Input Layer**: Takes in feature data (e.g., square footage, number of bedrooms).
* **Hidden Layers**: Processes the data through neurons.
* **Output Layer**: Produces a continuous output, which could be a single value (e.g., price).

To train an ANN for regression, the model adjusts its weights to minimize the difference between the predicted output and the actual target value using a loss function like Mean Squared Error (MSE). The training process involves backpropagation, where errors are propagated backward through the network, and weights are updated to reduce the loss.

**Key Steps in ANN Regression:**

1. **Data Preprocessing**: Normalization and scaling of data are crucial for ensuring efficient learning.
2. **Model Construction**: A neural network model is constructed with appropriate layers and activation functions.
3. **Training the Model**: The model is trained using a dataset, adjusting weights based on the error at each step.
4. **Evaluation**: The model’s performance is evaluated using metrics like MSE or Root Mean Squared Error (RMSE).

Python libraries like Keras and TensorFlow are often used to implement ANNs for regression, as they provide simple APIs for defining and training neural networks.

**Classification Problems with ANN**

In classification tasks, the goal is to categorize data into distinct classes or categories. For example, classifying images as either "cat" or "dog" or predicting whether a customer will buy a product based on their behavior are typical classification problems.

The structure of an ANN for classification is similar to that of regression, but with key differences in the output layer:

* **Input Layer**: Receives features of the data (e.g., image pixels, demographic features).
* **Hidden Layers**: Processes the data through neurons.
* **Output Layer**: Uses a softmax or sigmoid activation function to output class probabilities. In binary classification, the output layer consists of a single neuron with a sigmoid activation function. For multi-class classification, a softmax activation function is used.

For classification tasks, the model is trained using a loss function like **Cross-Entropy Loss**, which measures the difference between predicted probabilities and actual class labels. During training, the model learns to adjust its weights to reduce this loss.

**Key Steps in ANN Classification:**

**Data Preprocessing**: This involves encoding categorical variables (e.g., one-hot encoding for labels) and normalizing input features.

**Model Construction**: The model is designed with hidden layers and an output layer with appropriate activation functions (sigmoid for binary classification or softmax for multi-class).

**Training the Model**: The neural network is trained using a labeled dataset, and the weights are adjusted through backpropagation.

**Evaluation**: The performance is evaluated using accuracy, precision, recall, F1-score, or AUC (for binary classification).

**Challenges in ANN Models**

While ANNs are powerful, they come with several challenges:

1. **Overfitting**: ANN models, especially deep networks with many layers, can easily overfit the training data, leading to poor generalization. Regularization techniques such as dropout, L2 regularization, or early stopping can help mitigate overfitting.
2. **Data Quality**: ANN models are sensitive to the quality and quantity of data. Proper data preprocessing, including handling missing values, scaling, and feature engineering, is crucial.
3. **Model Tuning**: The performance of an ANN heavily depends on hyperparameters like the number of hidden layers, neurons, learning rate, and batch size. Grid search or random search methods are often employed to optimize these hyperparameters.
4. **Computational Complexity**: Training deep neural networks can be computationally expensive, requiring powerful hardware (e.g., GPUs) and careful resource management.

**Python Libraries for Implementing ANN**

Several Python libraries make it easy to implement ANN models for both regression and classification:

1. **TensorFlow/Keras**: These are the most popular frameworks for building and training neural networks. Keras provides a high-level API that is user-friendly, while TensorFlow provides more flexibility and control for advanced users.
2. **PyTorch**: Another powerful deep learning framework that offers flexibility and dynamic computation graphs, making it suitable for research and development.
3. **Scikit-learn**: Although primarily used for traditional machine learning algorithms, Scikit-learn provides simple neural network implementations (MLPRegressor and MLPClassifier) that are useful for smaller-scale tasks.

**Conclusion**

ANNs are versatile tools for both regression and classification tasks in machine learning. By leveraging their ability to model complex relationships and learn from data, they can achieve high accuracy in predicting continuous values and categorizing data into distinct classes. Python, with its rich ecosystem of libraries like Keras and TensorFlow, provides a powerful environment for implementing, training, and evaluating ANN models. However, successful deployment requires careful attention to data quality, model tuning, and computational resources to avoid common pitfalls like overfitting and high computational costs.

**Code:**

#Regression

import tensorflow as tf

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.datasets import fetch\_california\_housing

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

housing = fetch\_california\_housing()

df = pd.DataFrame(housing.data, columns=housing.feature\_names)

df['target'] = housing.target

df.head()

df=df.dropna()

X=df.iloc[:,:-1] ## independent features

y=df.iloc[:,-1] ## dependent features

X.isnull()

y.isnull()

sns.pairplot(df)

df.corr()

corrmat = df.corr()

top\_corr\_features = corrmat.index

plt.figure(figsize=(20,20))

g=sns.heatmap(df[top\_corr\_features].corr(),annot=True,cmap="RdYlGn")

corrmat.index

from sklearn.ensemble import ExtraTreesRegressor

import matplotlib.pyplot as plt

model = ExtraTreesRegressor()

model.fit(X,y)

X.head()

print(model.feature\_importances\_)

feat\_importances = pd.Series(model.feature\_importances\_, index=X.columns)

feat\_importances.nlargest(5).plot(kind='barh')

plt.show()

sns.distplot(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=0)

import keras

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import LeakyReLU,PReLU,ELU

from keras.layers import Dropout

NN\_model = Sequential()

# The Input Layer :

NN\_model.add(Dense(128, kernel\_initializer='normal',input\_dim = X\_train.shape[1], activation='relu'))

# The Hidden Layers :

NN\_model.add(Dense(256, kernel\_initializer='normal',activation='relu'))

NN\_model.add(Dense(256, kernel\_initializer='normal',activation='relu'))

NN\_model.add(Dense(256, kernel\_initializer='normal',activation='relu'))

# The Output Layer :

NN\_model.add(Dense(1, kernel\_initializer='normal',activation='linear'))

# Compile the network :

NN\_model.compile(loss='mean\_absolute\_error', optimizer='adam', metrics=['mean\_absolute\_error'])

NN\_model.summary()

# Fitting the ANN to the Training set

model\_history=NN\_model.fit(X\_train, y\_train,validation\_split=0.33, batch\_size = 10, epochs = 100)

prediction=NN\_model.predict(X\_test)

y\_test

sns.distplot(y\_test.values.reshape(-1,1)-prediction)

plt.scatter(y\_test,prediction)

from sklearn import metrics

print('MAE:', metrics.mean\_absolute\_error(y\_test, prediction))

print('MSE:', metrics.mean\_squared\_error(y\_test, prediction))

print('RMSE:', np.sqrt(metrics.mean\_squared\_error(y\_test, prediction)))

# Classification

import numpy as np

import pandas as pd

import tensorflow as tf

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.datasets import load\_iris

iris = load\_iris()

X = iris.data

y = iris.target

y = tf.keras.utils.to\_categorical(y, num\_classes=3)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

sc = StandardScaler()

X\_train = sc.fit\_transform(X\_train)

X\_test = sc.transform(X\_test)

ann = tf.keras.models.Sequential()

ann.add(tf.keras.layers.Dense(units=6,activation="relu"))

ann.add(tf.keras.layers.Dense(units=6,activation="relu"))

ann.add(tf.keras.layers.Dense(units=3,activation="sigmoid"))

ann.compile(optimizer="adam",loss="categorical\_crossentropy",metrics=['accuracy'])

ann.fit(X\_train,y\_train,batch\_size=32,epochs = 100)

loss, accuracy = ann.evaluate(X\_test, y\_test, verbose=0)

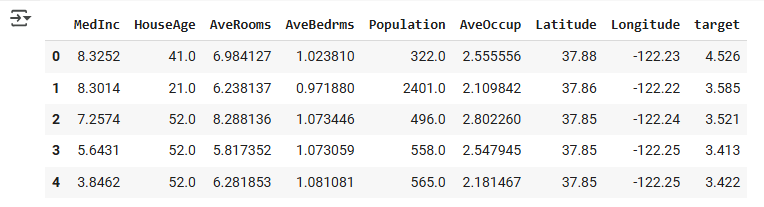
print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")

**Output:**

#Regression

df.head()



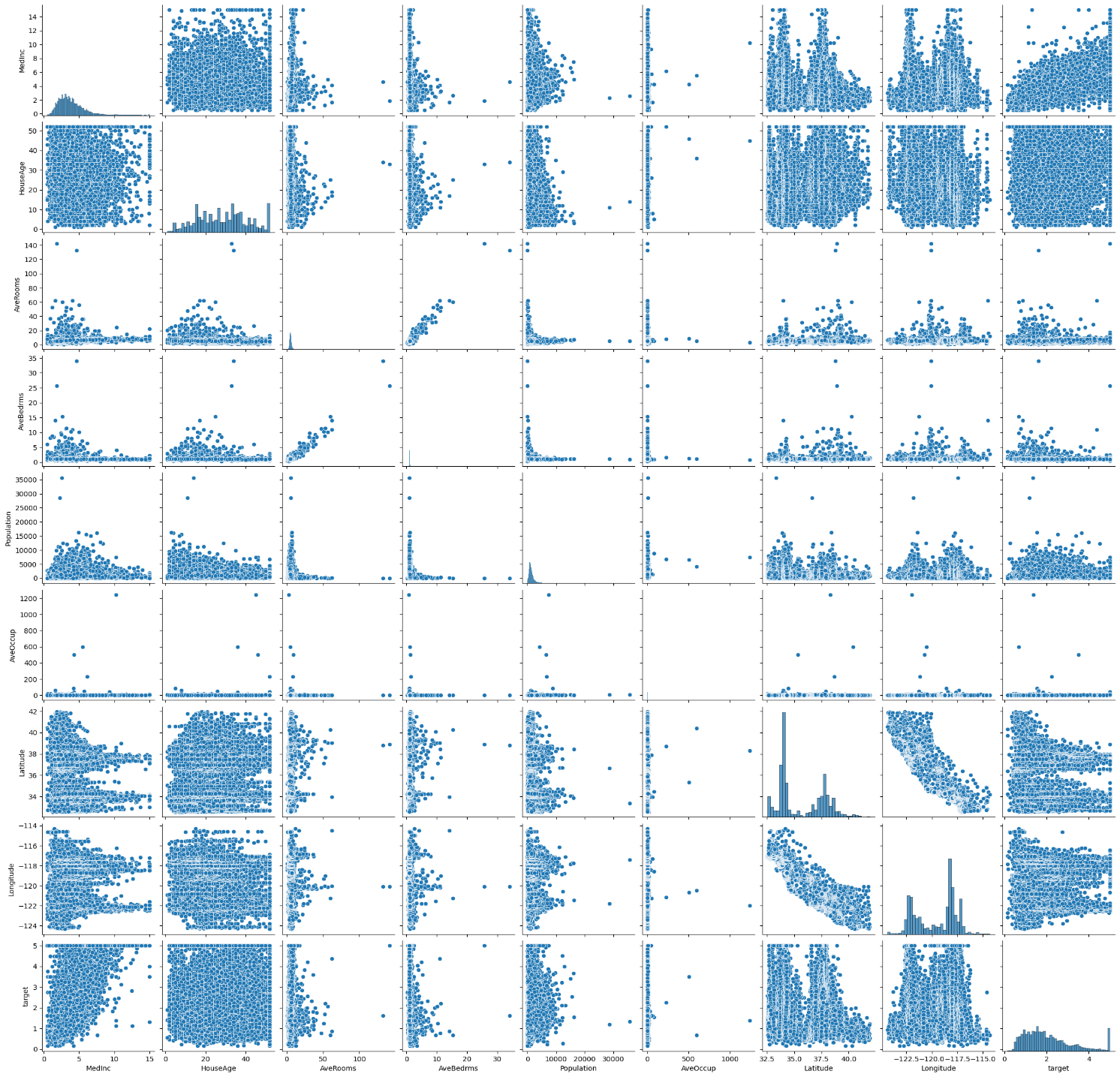
X.isnull()



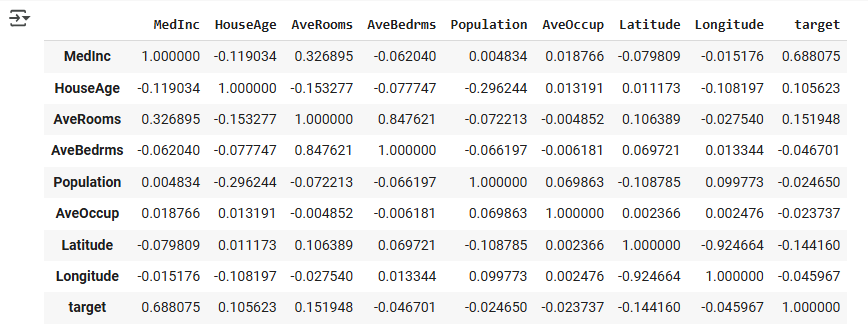
y.isnull()



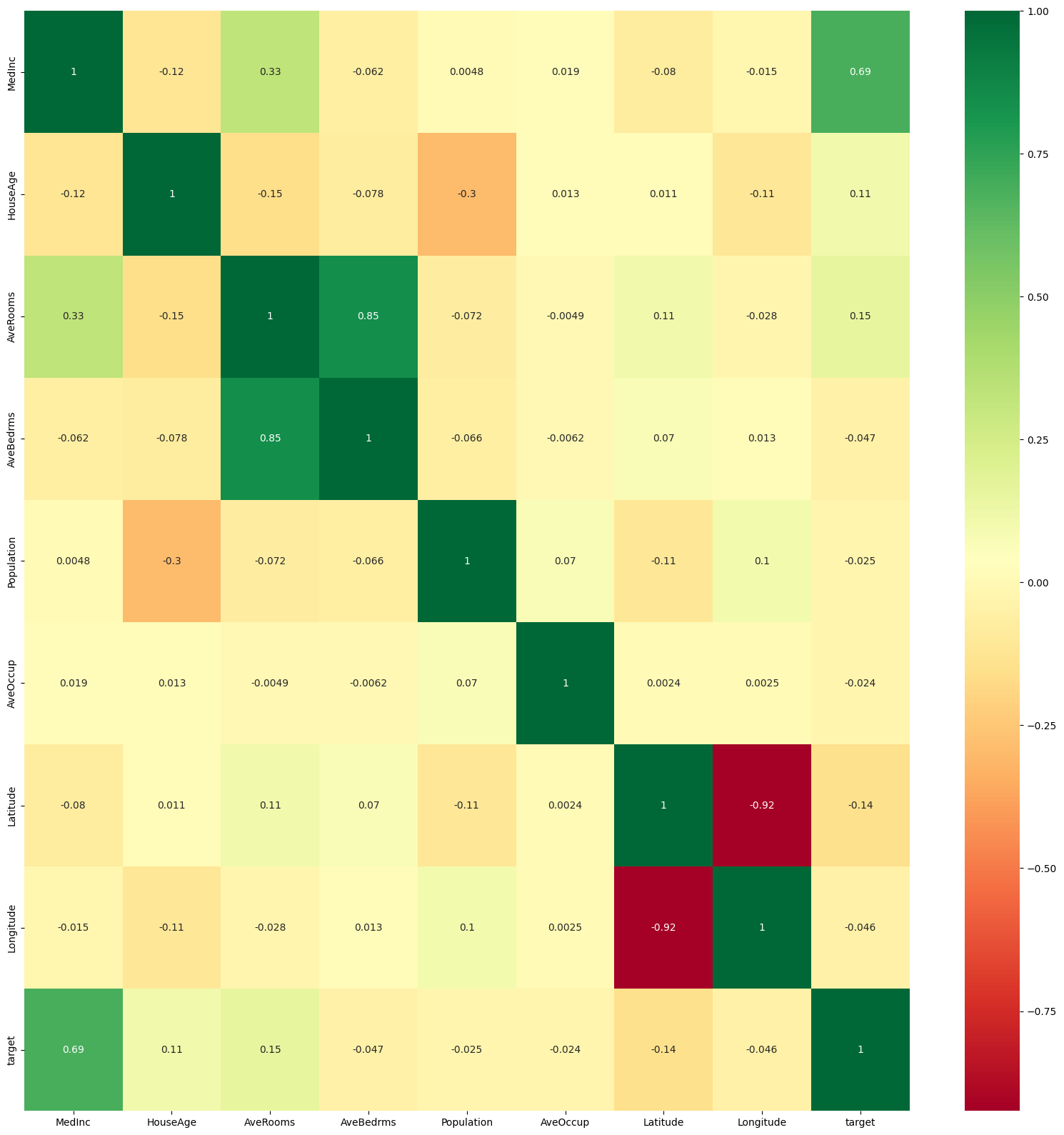
sns.pairplot(df)



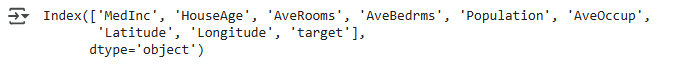
df.corr()



sns.heatmap()



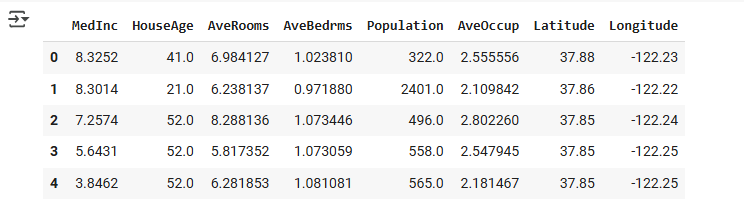
corrmat.index



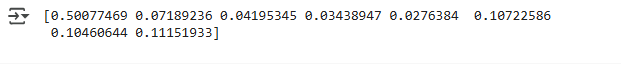
model.fit(X,y)



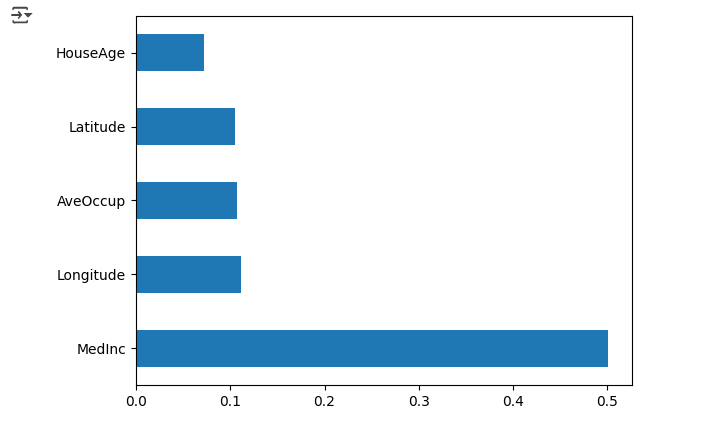
X.head()



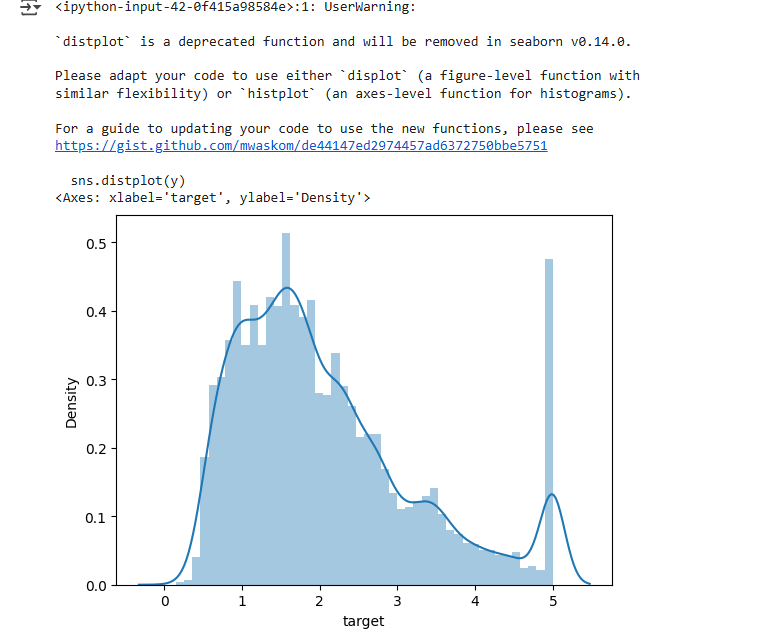
print(model.feature\_importances\_)



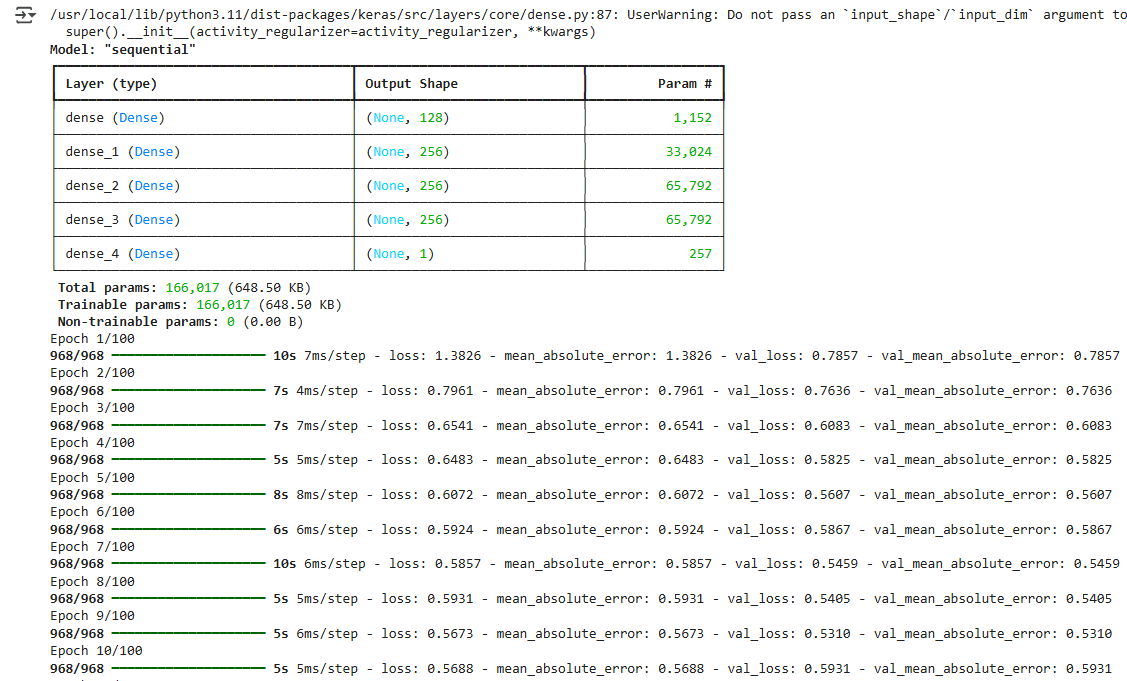
plt.show()



sns.distplot(y)



NN\_model.fit()



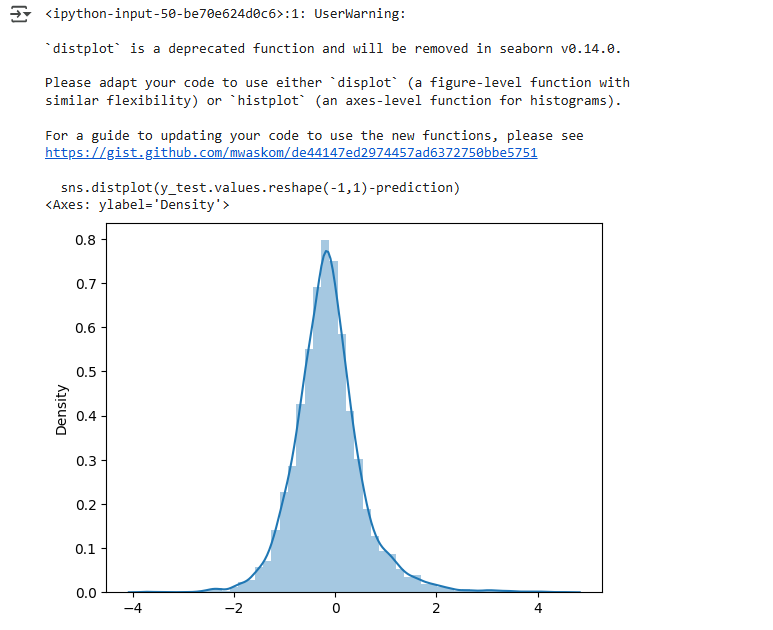
prediction=NN\_model.predict(X\_test)



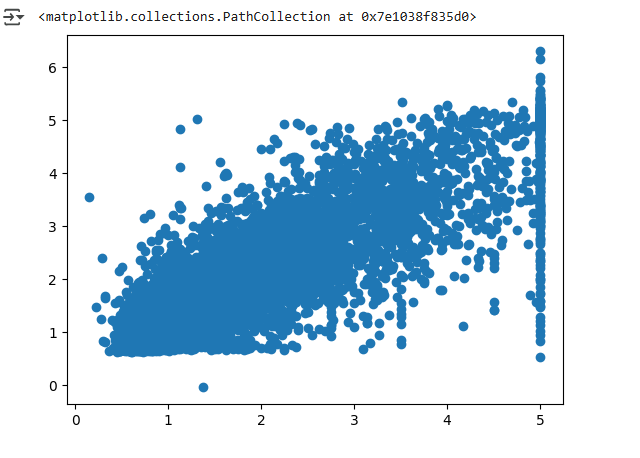
y\_test



sns.distplot(y\_test.values.reshape(-1,1)-prediction)



plt.scatter(y\_test,prediction)



# MAE, MSE and RMSE

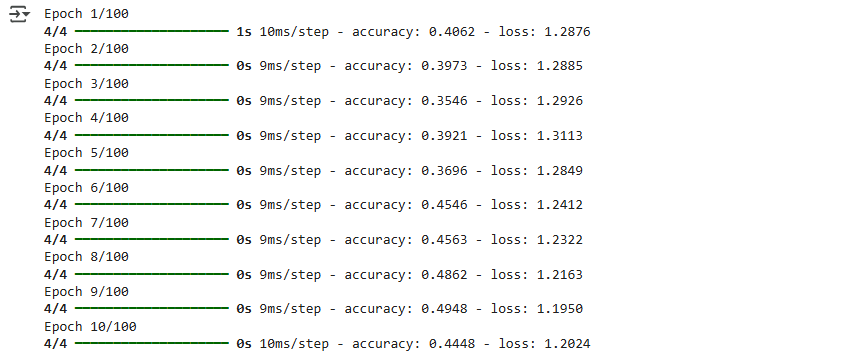


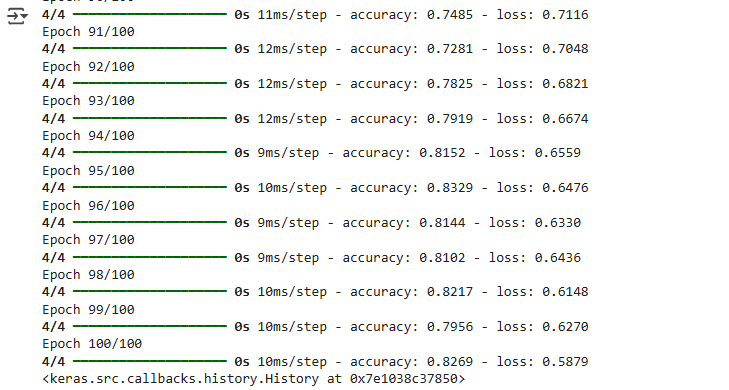
#Classification

Shape of : X\_train , y\_train



ann.fit()





#Loss and accuracy



**Learning Outcomes:**

* Gain the ability to build and train ANN models for both regression and classification tasks in Python.
* Learn to preprocess data and evaluate model performance using key metrics like MSE, accuracy, and F1-score.

**Experiment 3**

## **Aim: Implementation of Convolution Neural Network for MRI Data Set in Python.**

**Objectives:**

* To implement CNN model in Python on MRI images.
* To implement CNN model for classifying images and evaluate its performance using performance metrics.

**Theory:**

**Introduction to Deep Learning and CNNs**

Deep learning is a subset of machine learning that focuses on training artificial neural networks to learn from large amounts of data. Among deep learning architectures, **Convolutional Neural Networks (CNNs)** are particularly effective for analyzing image-based data, making them suitable for medical imaging tasks such as MRI (Magnetic Resonance Imaging) analysis.

MRI scans provide detailed internal body structures, helping in medical diagnosis, treatment planning, and research. However, manually interpreting MRI images is time-consuming and subject to variability. CNNs help automate and improve MRI analysis by detecting patterns, classifying abnormalities, and segmenting regions of interest.

**Convolutional Neural Networks (CNNs) in Medical Imaging**

CNNs are specialized deep learning models designed for image processing tasks. They use **convolutional layers** to extract spatial features from images, making them efficient in identifying patterns and structures within MRI scans. The fundamental components of a CNN include:

* **Convolutional Layers:** Apply filters to input images to extract features such as edges, textures, and shapes.
* **Pooling Layers:** Reduce the dimensionality of the data while preserving important features, improving computational efficiency.
* **Fully Connected Layers:** Flatten the extracted features and connect them to output neurons for classification or segmentation tasks.
* **Activation Functions (ReLU, Softmax, Sigmoid):** Introduce non-linearity into the model to improve learning capability.

CNNs have significantly advanced medical imaging applications, enabling automated tumor detection, brain segmentation, and disease classification from MRI datasets.

**MRI Dataset and Its Importance**

MRI datasets contain grayscale images that represent different tissues and structures within the human body. These datasets are crucial for training CNN models in various medical applications, such as:

* **Brain Tumor Classification** (Normal vs. Tumor)
* **Alzheimer’s Disease Detection**
* **Organ Segmentation**

MRI datasets usually require preprocessing steps, including normalization, resizing, and augmentation, to improve model performance.

**Implementation of CNN for MRI Dataset in Python**

Python, with deep learning libraries such as **TensorFlow** and **Keras**, provides an efficient framework for implementing CNNs. The basic steps involved in developing a CNN for MRI image classification include:

1. **Loading the MRI Dataset:** Using libraries like Pandas, NumPy, or TensorFlow datasets.
2. **Preprocessing the Data:** Resizing images, normalizing pixel values, and augmenting the dataset.
3. **Building the CNN Model:** Defining layers such as convolutional, pooling, and fully connected layers.
4. **Compiling and Training the Model:** Using an optimizer (SGD, Adam) and loss function (categorical cross-entropy).
5. **Evaluating the Model:** Testing the model on unseen MRI images and assessing its accuracy.

**Applications of CNNs in MRI Analysis**

* **Brain Tumor Detection:** CNNs can classify MRI images as normal or containing tumors with high accuracy.
* **Alzheimer’s Disease Prediction:** Identifying structural changes in brain MRI scans to diagnose neurodegenerative diseases.
* **Medical Image Segmentation:** Extracting regions of interest, such as tumors or organs, from MRI scans for precise analysis.

**Conclusion**

The implementation of a Convolutional Neural Network (CNN) for MRI image classification is a powerful deep learning application in medical imaging. CNNs efficiently learn spatial hierarchies of features and can significantly improve disease detection accuracy. With Python-based frameworks like TensorFlow and Keras, researchers and medical professionals can develop robust models to analyze MRI scans, contributing to advancements in medical diagnostics and healthcare.

**Code:**

import pandas as pd

import numpy as np

import cv2

import os

import random

import matplotlib.pyplot as plt

import seaborn as sns

from matplotlib.image import imread

import tensorflow as tf

import tensorflow\_hub as hub

os.environ['TF\_CPP\_MIN\_LOG\_LEVEL'] = '3'

from keras.models import Sequential

from keras.layers import Conv2D,Activation,MaxPooling2D,Dense,Flatten, Dropout

from keras.losses import BinaryCrossentropy

try:

    from keras.optimizer import Adam

except:

    from tensorflow.keras.optimizers import Adam

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import auc, roc\_curve, log\_loss, roc\_auc\_score, auc, roc\_curve, accuracy\_score, confusion\_matrix, cohen\_kappa\_score

import warnings

warnings.filterwarnings('ignore')

MAIN\_DIR = "/content/brain\_tumor\_dataset/"

# control reproducibility with random seeds

seed\_value = 1

np.random.seed(seed\_value)

random.seed(seed\_value)

tf.random.set\_seed(seed\_value)

# Define location of dataset

folder\_no = MAIN\_DIR + '/no'

folder\_yes = MAIN\_DIR + '/yes'

# Collect image paths

image\_paths = []

for filepath in [folder\_no, folder\_yes]:

    for f in os.listdir(filepath)[:3]:  # Take the first 3 images per folder

        image\_paths.append(os.path.join(filepath, f))

fig, axes = plt.subplots(1, len(image\_paths), figsize=(15, 5))

for ax, img\_path in zip(axes, image\_paths):

    image = imread(img\_path)

    ax.imshow(image)

plt.tight\_layout()

plt.show()

filepath = MAIN\_DIR

images, labels, filenames = [], [], []

for gt in ['yes','no']:

    filepath\_gt = filepath + gt

    for f in os.listdir(filepath\_gt):

        filename = filepath\_gt + '/' + f

        if '.\_' not in filename: # metadata files created by macos

            # load image

            photo = tf.keras.utils.load\_img(path=filename,target\_size=(200, 200))

            # load image pixels

            image = imread(filename)

            resized\_image = cv2.resize(image,dsize=(200,200))

            # in case of grayscale images

            if len(np.shape(resized\_image)) > 2:

                # convert the image from COLOR\_BGR2GRAY

                resized\_image = cv2.cvtColor(resized\_image, cv2.COLOR\_BGR2GRAY)

            # store to array

            images.append(resized\_image)

            filenames.append(filename)

            if gt=='yes':

                label = 1

                labels.append(label)

            elif gt=='no':

                label = 0

                labels.append(label)

gtFr = pd.DataFrame(labels).rename(columns={0:'gt'}).reset\_index(drop=False).rename(columns={'index':'pID'})

gtFr['filename'] = filenames

# split into training and testing sets, stratifying by gt for equal representation

trainFr, testFr = train\_test\_split(gtFr, test\_size=0.2, stratify=gtFr['gt'])

# store training/testing indices

trainFr['set'] = 'train'

testFr['set'] = 'test'

gtFr2 = pd.concat([trainFr,testFr],axis=0)

gtFr2.set\_index(['filename']).to\_csv('base\_model\_train\_test.csv')

files\_train = []

X\_train = []

y\_train = []

for idx, row in trainFr.iterrows():

    # load image pixels

    image = imread(row['filename'])

    # resize image to standard 200x200

    resized\_image = cv2.resize(image,dsize=(200,200))

    # in case of grayscale images

    if len(np.shape(resized\_image)) > 2:

        # convert the image from COLOR\_BGR2GRAY

        resized\_image = cv2.cvtColor(resized\_image, cv2.COLOR\_BGR2GRAY)

    # store to array

    X\_train.append(resized\_image)

    y\_train.append(row['gt'])

    files\_train.append(row['filename'])

# reshape data to fit model

X\_train = np.array(X\_train).reshape(len(trainFr),200,200,1)

y\_train = np.array(y\_train)

files\_test = []

X\_test = []

y\_test = []

for idx, row in testFr.iterrows():

    # load image pixels

    image = imread(row['filename'])

    # resize image to standard 200x200

    resized\_image = cv2.resize(image,dsize=(200,200))

    # in case of grayscale images

    if len(np.shape(resized\_image)) > 2:

        # convert the image from COLOR\_BGR2GRAY

        resized\_image = cv2.cvtColor(resized\_image, cv2.COLOR\_BGR2GRAY)

    # store to array

    X\_test.append(resized\_image)

    y\_test.append(row['gt'])

    files\_test.append(row['filename'])

# reshape data to fit model

X\_test = np.array(X\_test).reshape(len(testFr),200,200,1)

y\_test = np.array(y\_test)

# create model

model = Sequential()

# add convolutional layer

model.add(Conv2D(64, kernel\_size=3, activation='relu', input\_shape=(200,200,1))) # input is of shape (N, C) or (N, C, L) where N is the batch size as before. However what does the C and L denote here? It seems that C = number of features, L = number of channels,

# add max pooling layer

model.add(MaxPooling2D(pool\_size=(2, 2)))

# add another convolutional layer

model.add(Conv2D(32, kernel\_size=3, activation='relu'))

# add another max pooling layer

model.add(MaxPooling2D(pool\_size=(2, 2)))

# flatten output (connect convolutional layers and dense layers)

model.add(Flatten())

# add a dense layer with 128 neurons and ReLU activation

model.add(Dense(128, activation='relu'))

# add dense layer

model.add(Dense(1, activation='sigmoid'))

# compile model using accuracy to measure model performance

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

history = model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=30)

# Function to plot loss and accuracy in a single row

def plot\_curves(history):

    """

    Returns loss and accuracy curves in a single row.

    """

    loss = history.history["loss"]

    val\_loss = history.history["val\_loss"]

    accuracy = history.history["accuracy"]

    val\_accuracy = history.history["val\_accuracy"]

    epochs = range(len(loss))

    # Create a single row with 2 plots (loss and accuracy)

    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    # Plot loss

    axes[0].plot(epochs, loss, label="training\_loss")

    axes[0].plot(epochs, val\_loss, label="val\_loss")

    axes[0].set\_title("Loss")

    axes[0].set\_xlabel("Epochs")

    axes[0].legend()

    # Plot accuracy

    axes[1].plot(epochs, accuracy, label="training\_accuracy")

    axes[1].plot(epochs, val\_accuracy, label="val\_accuracy")

    axes[1].set\_title("Accuracy")

    axes[1].set\_xlabel("Epochs")

    axes[1].legend()

    plt.tight\_layout()  # Adjust layout for better spacing

    plt.show()

plot\_curves(history)

# make predictions

predictions = model.predict(X\_test)

# store as probabilities

probabilities = [p[0] for p in predictions]

testFr['pred'] = probabilities

testFr.set\_index(['pID']).to\_csv('base\_model\_predictions.csv')

dataPos = testFr[testFr['gt']==1]['pred'].values

dataNeg = testFr[testFr['gt']==0]['pred'].values

dataAll = np.concatenate((dataPos, dataNeg))

lblArr = np.zeros(len(dataAll), dtype=bool)

lblArr[0:len(dataPos)] = True

fpr, tpr, thresholds = roc\_curve(lblArr, dataAll, pos\_label=True)

roc\_auc = auc(fpr, tpr)

# invert comparison if (ROC<0.5) required

if roc\_auc<0.5:

    lblArr = ~lblArr

    fpr, tpr, thresholds = roc\_curve(lblArr, dataAll, pos\_label=True)

    roc\_auc = auc(fpr, tpr)

    print('inverting labels')

print('ROC AUC: {:0.2f}'.format(roc\_auc))

# calculate best cut-off based on distance to top corner of ROC curve

distArr = np.sqrt(np.power(fpr, 2) + np.power((1 - tpr), 2))

cutoffIdx = np.argsort(distArr)[0]

cutoffTh = thresholds[cutoffIdx]

print('Cutoff threshold that maximizes sens/spec: {:0.2f}'.format(cutoffTh))

lblOut = dataAll >= cutoffTh

acc = accuracy\_score(lblArr, lblOut)

print('Accuracy: {:0.2f}'.format(acc))

sens = tpr[cutoffIdx]

print('Sensitivity: {:0.2f}'.format(sens))

spec = 1 - fpr[cutoffIdx]

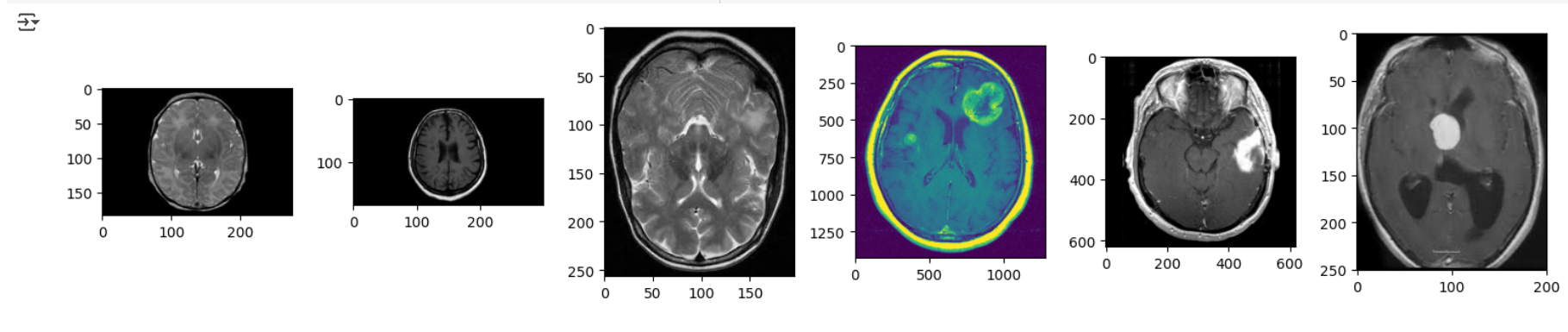
print('Specificity: {:0.2f}'.format(spec))

kappa = cohen\_kappa\_score(lblOut, lblArr)

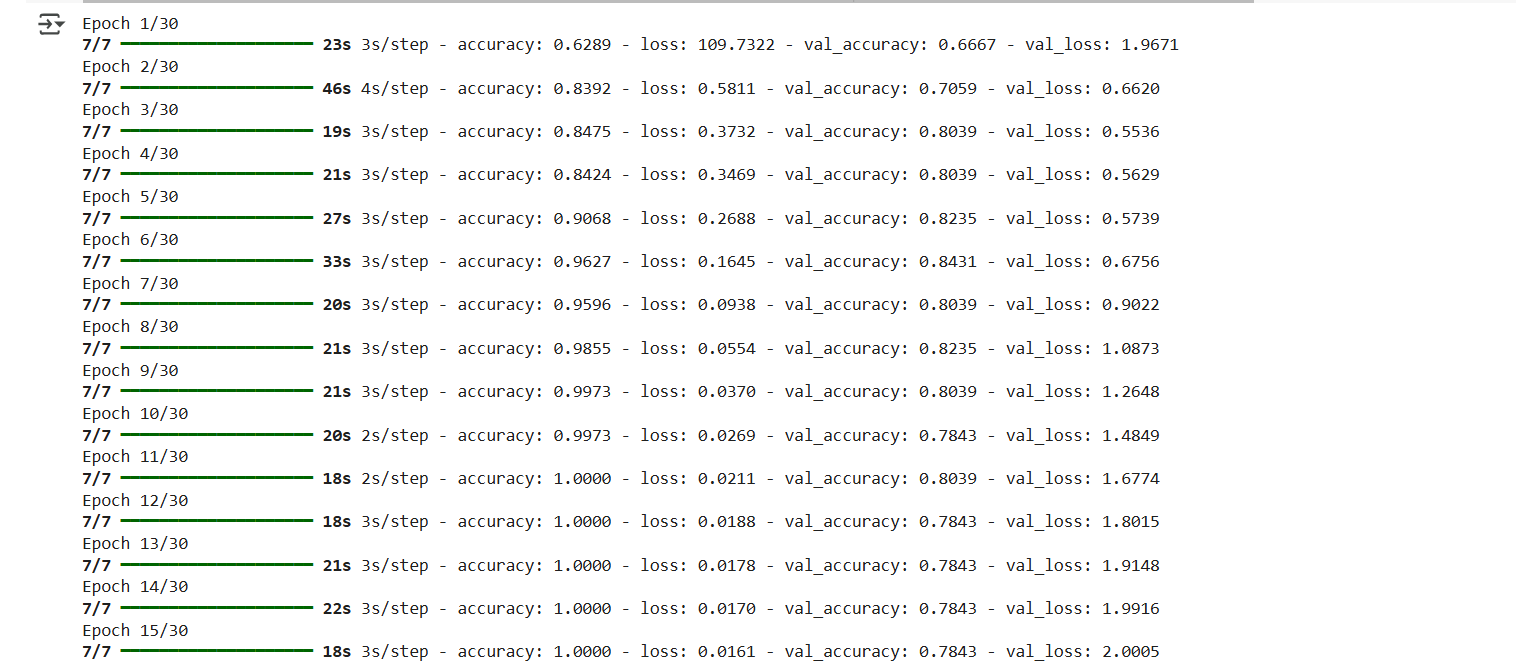
print('Cohen kappa score: {:0.2f}'.format(kappa))

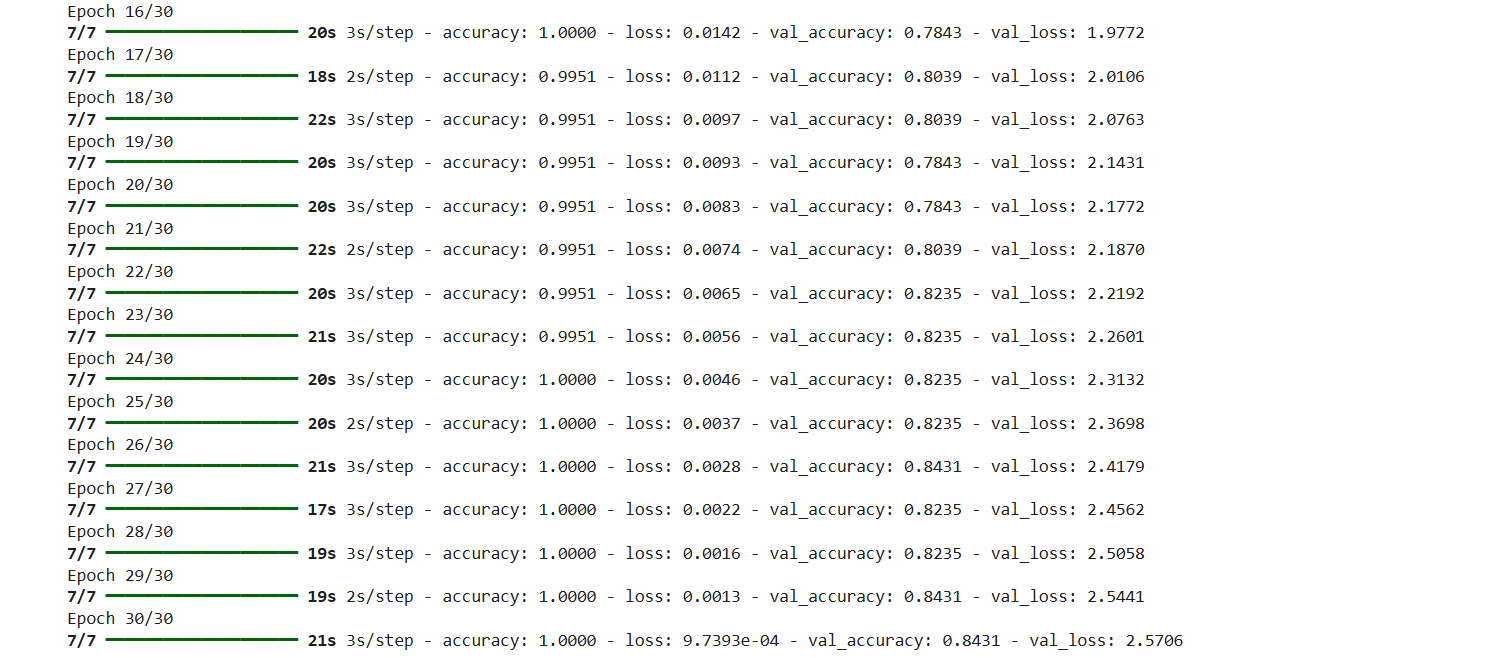
**Output:**

# First 3 images per folder

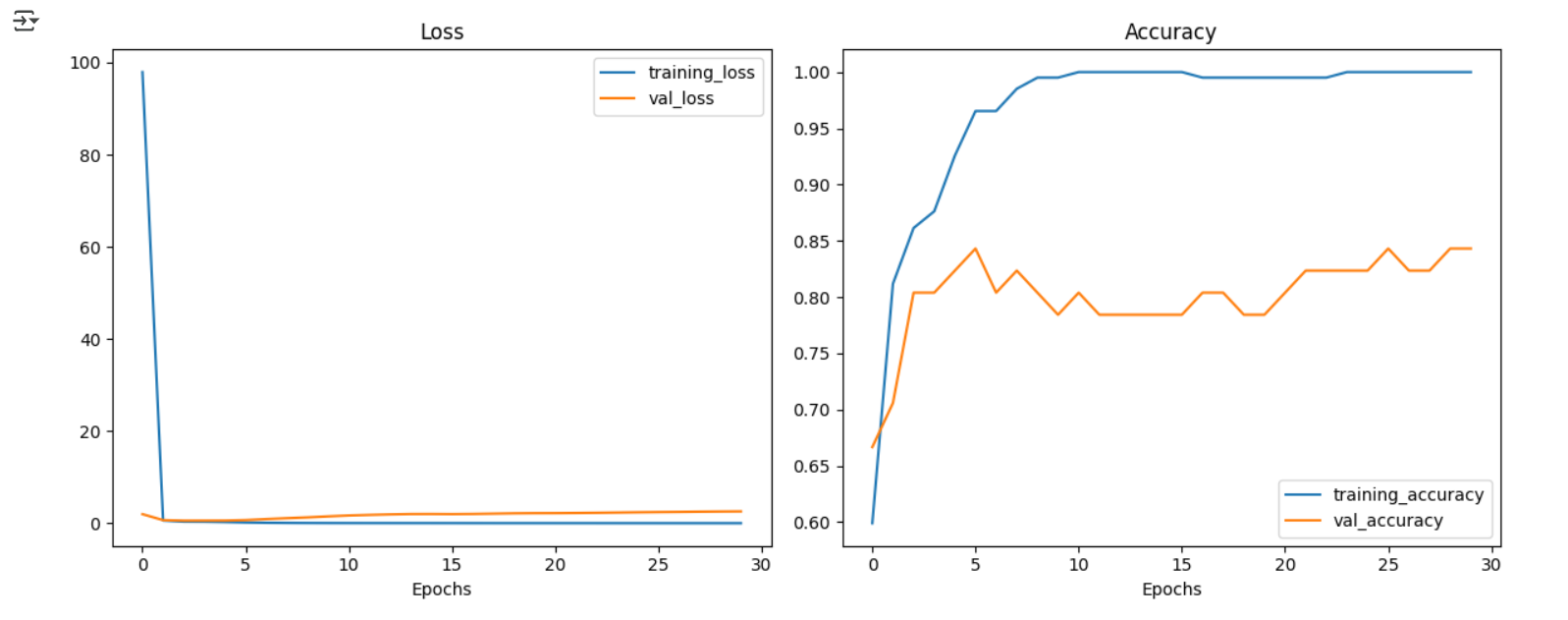


model.fit()

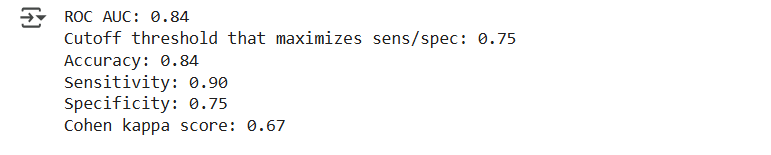




plot\_curves(history)



# Evaluation Metrics



**Learning Outcomes:**

Gain the ability to build and train CNN models for classification tasks on images in Python.

Learnt to preprocess and scale image data to train CNN model and evaluate model performance using key metrics like accuracy, sensitivity and kappa score.

**Experiment 4**

**Aim: Implementation of Autoencoders for dimensionality reduction in Python.**

**Objectives:**

* To implement Autoencoders in Python for dimensionality reduction.
* To train an Autoencoder using TensorFlow and Keras and analyze its reconstruction ability.

**Theory:**

**Introduction to Deep Learning and Dimensionality Reduction**

Deep learning is a subset of machine learning that enables computers to learn from large datasets using artificial neural networks. One of the critical challenges in machine learning is dealing with **high-dimensional data**, which can lead to increased computational costs and reduced model performance due to the "curse of dimensionality." **Dimensionality reduction** is a technique used to transform high-dimensional data into a lower-dimensional representation while preserving important features.

Among various dimensionality reduction techniques, **Autoencoders** have gained significant importance. Autoencoders are a type of neural network architecture used to learn efficient data representations in an **unsupervised manner**. They compress input data into a **lower-dimensional latent space** and then reconstruct it back, capturing the most critical features of the data.

**What are Autoencoders?**

An **Autoencoder (AE)** is a neural network designed to encode input data into a lower-dimensional space (encoding) and then reconstruct the original input (decoding). It consists of two main components:

1. **Encoder:** Reduces the input dimensions by learning compressed representations.
2. **Decoder:** Reconstructs the input data from the compressed representation.

The objective of an autoencoder is to minimize the **reconstruction loss**, typically measured using **Mean Squared Error (MSE)** or **Binary Cross-Entropy Loss**.

**Applications of Autoencoders in Dimensionality Reduction**

Autoencoders are widely used for dimensionality reduction and feature extraction, especially in cases where linear methods like PCA (Principal Component Analysis) are insufficient. Some key applications include:

* **Data Compression:** Efficiently encoding high-dimensional data into a compact representation.
* **Noise Reduction (Denoising Autoencoders):** Removing noise from images or signals by learning a cleaner representation.
* **Anomaly Detection:** Identifying rare patterns by reconstructing normal data patterns and detecting deviations.
* **Feature Extraction:** Learning meaningful representations for downstream machine learning tasks.

**Implementation of Autoencoders for Dimensionality Reduction in Python**

Autoencoders can be implemented using **TensorFlow** and **Keras** in Python. The basic steps include:

1. **Loading the Dataset:** Common datasets such as **MNIST** or **CIFAR-10** are used to train autoencoders.
2. **Preprocessing the Data:** Normalization and reshaping of input images.
3. **Building the Autoencoder Model:**
   * A simple autoencoder consists of an **encoder (Dense layers)** followed by a **decoder (Dense layers)**.
4. **Training the Autoencoder:** Using an **Adam optimizer** and **Mean Squared Error (MSE) loss**.
5. **Evaluating the Performance:** Checking the quality of reconstructed images and analyzing dimensionality reduction.

**Advantages of Autoencoders for Dimensionality Reduction**

Compared to traditional techniques like PCA, autoencoders offer several benefits:

* **Non-linearity Handling:** Autoencoders can learn complex, non-linear relationships in data.
* **Better Representation Learning:** Captures hierarchical and meaningful features.
* **Scalability:** Works efficiently on large datasets without significant performance degradation.
* **Feature Learning without Labels:** Unlike supervised learning, autoencoders require no labeled data.

**Conclusion**

Autoencoders provide an efficient way to perform **dimensionality reduction**, especially for high-dimensional datasets. They are widely used in feature extraction, denoising, and anomaly detection tasks. With deep learning frameworks like **TensorFlow and Keras**, implementing autoencoders in Python has become straightforward. Their ability to learn efficient representations without supervision makes them a powerful tool in modern machine learning and deep learning applications.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, LeakyReLU

from tensorflow.keras.optimizers import RMSprop

from sklearn.manifold import TSNE

# Load Dataset

file\_path = "train.csv"  # Change if needed

df = pd.read\_csv(file\_path)

print(df.head())

# EDA

print(df.info())

print(df.describe())

print(df.isnull().sum().sum(), "missing values")

# Distribution of Target Variable

sns.histplot(df['target'], bins=30, kde=True)

plt.title("Target Variable Distribution")

plt.show()

# Remove ID and extract features

df.drop(columns=['ID\_code'], inplace=True)

X = df.drop(columns=['target']).values  # Features

y = df['target'].values  # Target

# Normalize Data

scaler = StandardScaler()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Handle NaN or Inf values

X\_train = np.nan\_to\_num(X\_train)

X\_test = np.nan\_to\_num(X\_test)

# Autoencoder Architecture

encoding\_dim = 10  # Dimension reduction size

input\_dim = X\_train.shape[1]

input\_layer = Input(shape=(input\_dim,))

encoded = Dense(64)(input\_layer)

encoded = LeakyReLU(alpha=0.1)(encoded)

encoded = Dense(32)(encoded)

encoded = LeakyReLU(alpha=0.1)(encoded)

encoded = Dense(encoding\_dim, activation='linear')(encoded)

decoded = Dense(32)(encoded)

decoded = LeakyReLU(alpha=0.1)(decoded)

decoded = Dense(64)(decoded)

decoded = LeakyReLU(alpha=0.1)(decoded)

decoded = Dense(input\_dim, activation='linear')(decoded)

# Define Autoencoder

autoencoder = Model(input\_layer, decoded)

autoencoder.compile(optimizer=RMSprop(learning\_rate=0.0001), loss='mae')

# Train Autoencoder

autoencoder.fit(X\_train, X\_train, epochs=50, batch\_size=256, shuffle=True, validation\_data=(X\_test, X\_test))

# Extract Encoder

encoder = Model(input\_layer, encoded)

X\_train\_encoded = encoder.predict(X\_train)

X\_test\_encoded = encoder.predict(X\_test)

# t-SNE Visualization

tsne = TSNE(n\_components=2, random\_state=42)

X\_embedded = tsne.fit\_transform(X\_test\_encoded)

plt.scatter(X\_embedded[:, 0], X\_embedded[:, 1], c=y\_test[:len(X\_embedded)], cmap='coolwarm', alpha=0.5)

plt.colorbar()

plt.title("t-SNE Visualization of Encoded Features")

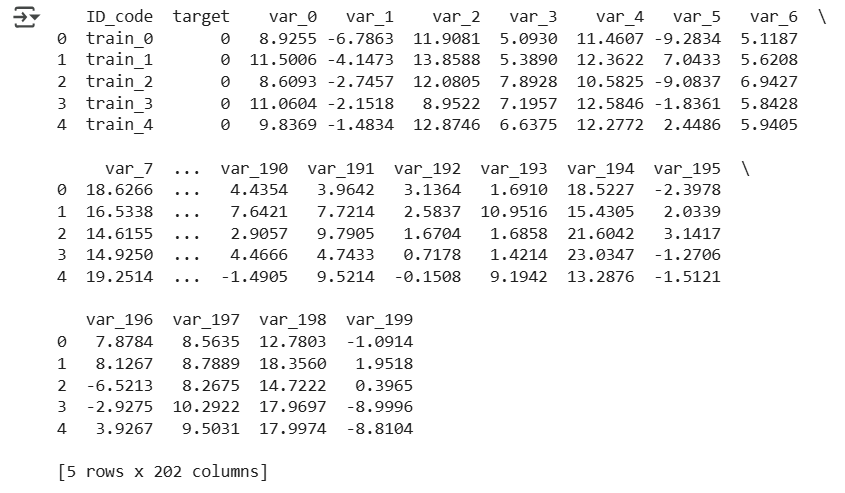
plt.show()

print("Original feature shape:", X\_train.shape)

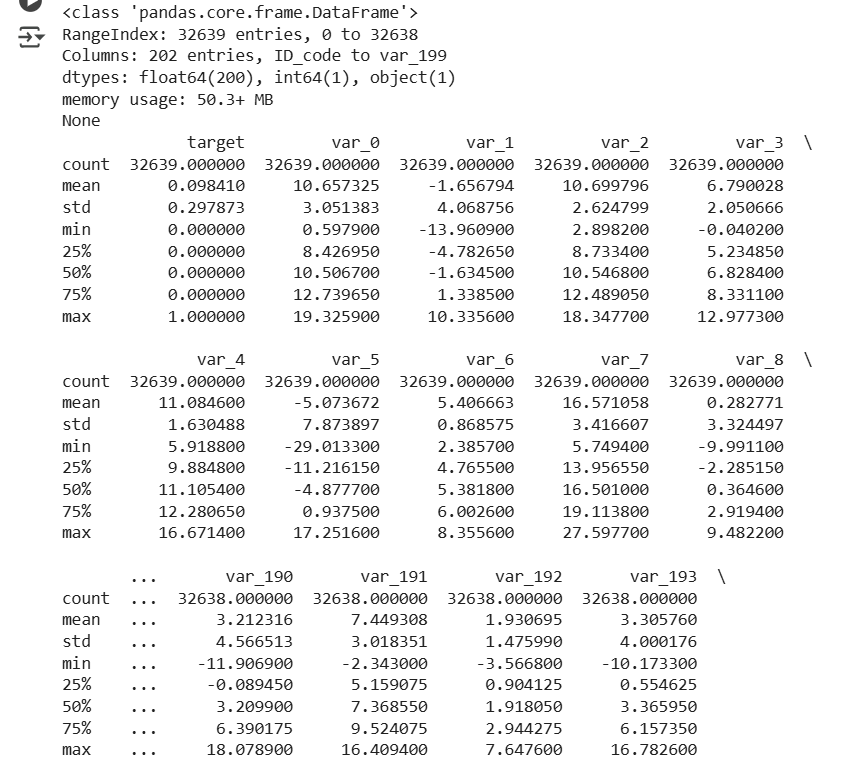
print("Reduced feature shape:", X\_train\_encoded.shape)

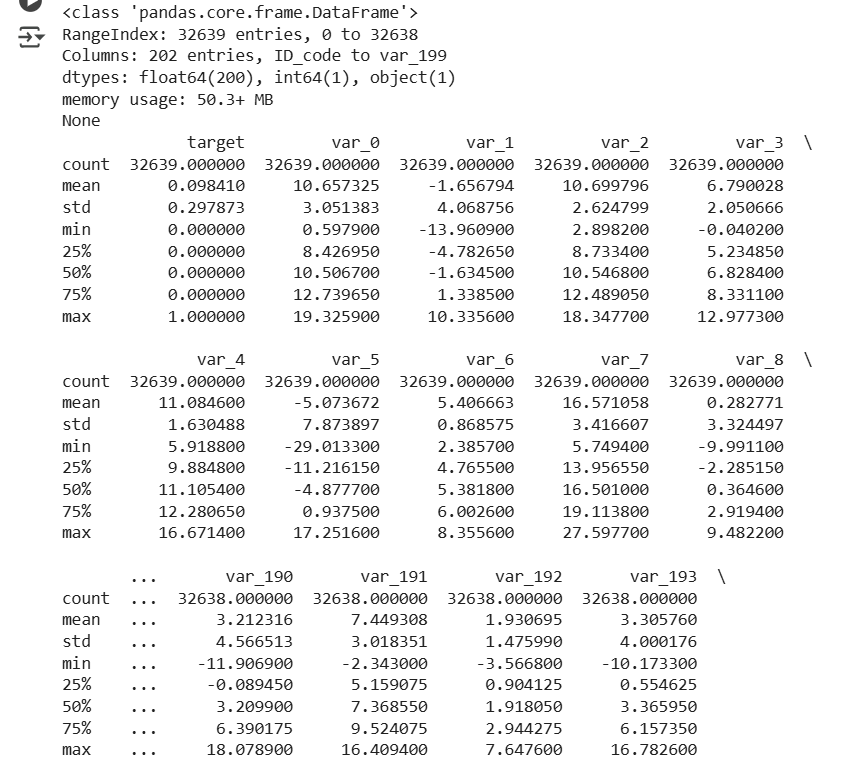
**Output:**

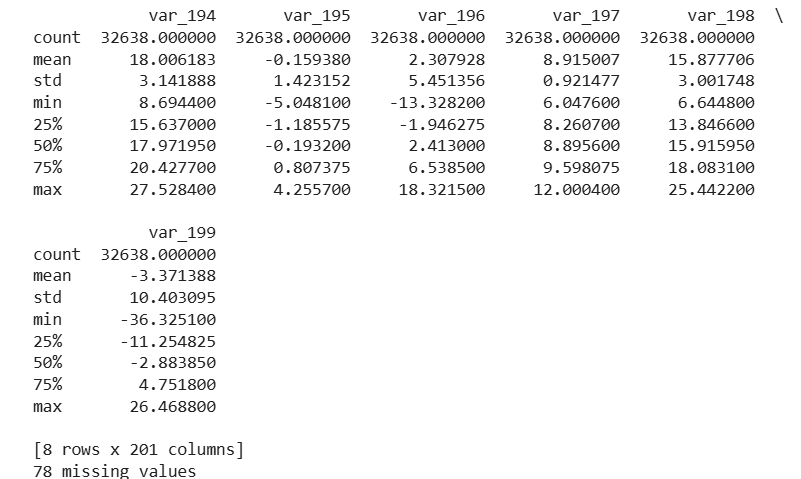
# Load dataset - df.head()



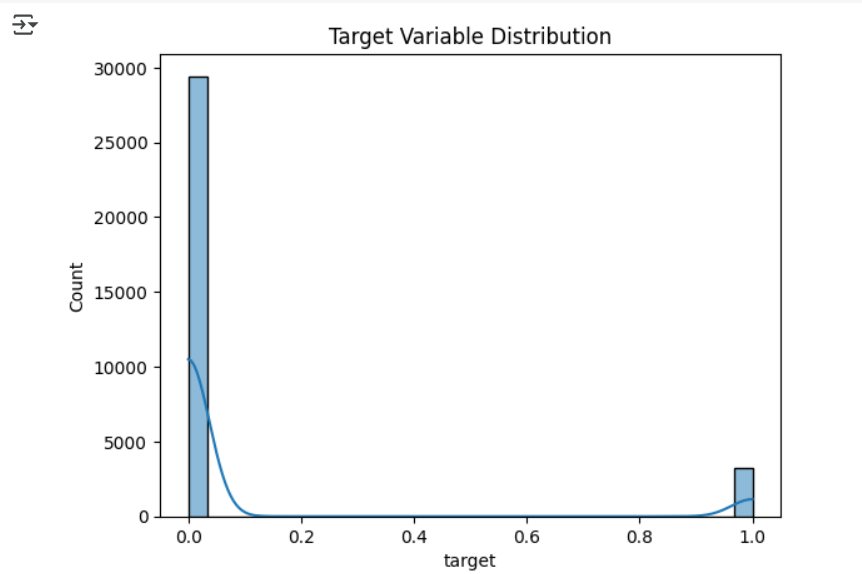
# EDA



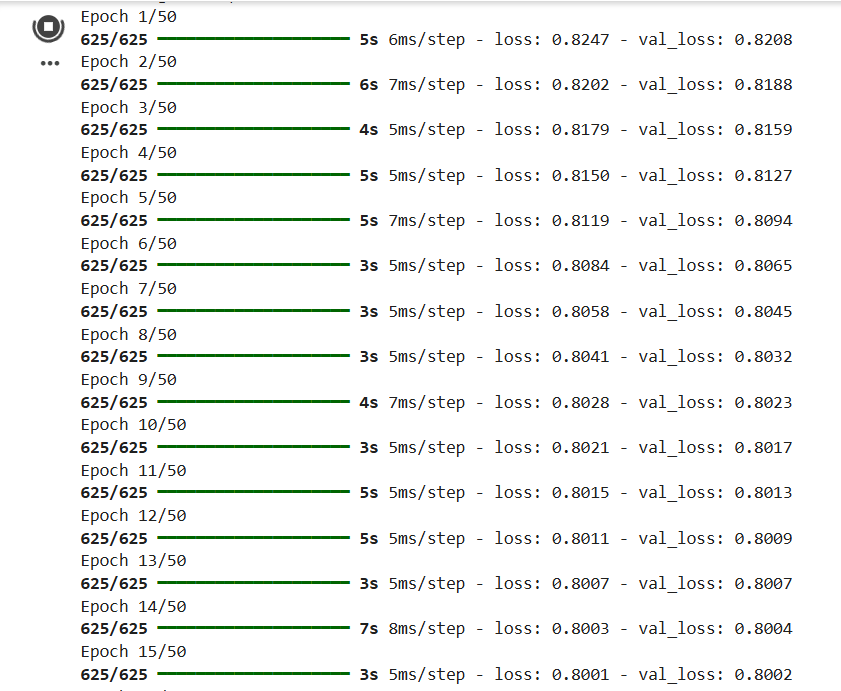




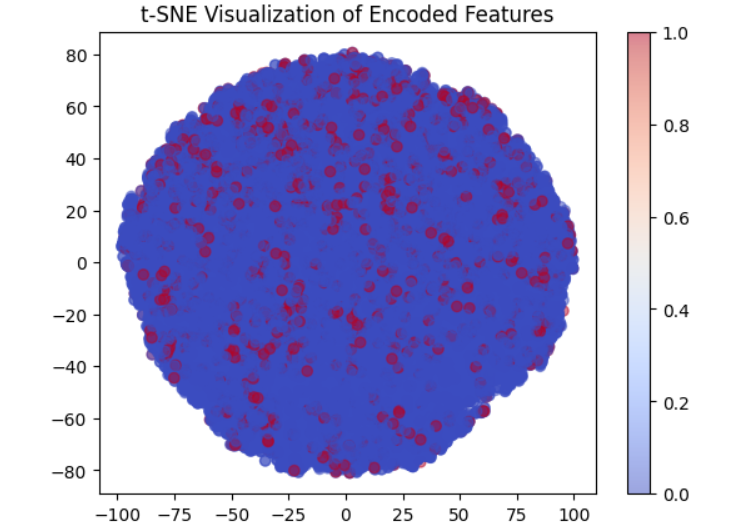
# Distribution of Target Variable



# autoencoder.fit()



# t-SNE Visualization



# Original vs Reduced



**Learning Outcomes:**

Understand the concept of autoencoders and their role in dimensionality reduction.

Learn to preprocess and normalize high-dimensional data for efficient training.

**Experiment 4**

**Aim: Implementation of Autoencoders for dimensionality reduction in Python.**

**Objectives:**

* To implement Autoencoders in Python for dimensionality reduction.
* To train an Autoencoder using TensorFlow and Keras and analyze its reconstruction ability.

**Theory:**

**Introduction to Deep Learning and Dimensionality Reduction**

Deep learning is a subset of machine learning that enables computers to learn from large datasets using artificial neural networks. One of the critical challenges in machine learning is dealing with **high-dimensional data**, which can lead to increased computational costs and reduced model performance due to the "curse of dimensionality." **Dimensionality reduction** is a technique used to transform high-dimensional data into a lower-dimensional representation while preserving important features.

Among various dimensionality reduction techniques, **Autoencoders** have gained significant importance. Autoencoders are a type of neural network architecture used to learn efficient data representations in an **unsupervised manner**. They compress input data into a **lower-dimensional latent space** and then reconstruct it back, capturing the most critical features of the data.

**What are Autoencoders?**

An **Autoencoder (AE)** is a neural network designed to encode input data into a lower-dimensional space (encoding) and then reconstruct the original input (decoding). It consists of two main components:

1. **Encoder:** Reduces the input dimensions by learning compressed representations.
2. **Decoder:** Reconstructs the input data from the compressed representation.

The objective of an autoencoder is to minimize the **reconstruction loss**, typically measured using **Mean Squared Error (MSE)** or **Binary Cross-Entropy Loss**.

**Applications of Autoencoders in Dimensionality Reduction**

Autoencoders are widely used for dimensionality reduction and feature extraction, especially in cases where linear methods like PCA (Principal Component Analysis) are insufficient. Some key applications include:

* **Data Compression:** Efficiently encoding high-dimensional data into a compact representation.
* **Noise Reduction (Denoising Autoencoders):** Removing noise from images or signals by learning a cleaner representation.
* **Anomaly Detection:** Identifying rare patterns by reconstructing normal data patterns and detecting deviations.
* **Feature Extraction:** Learning meaningful representations for downstream machine learning tasks.

**Implementation of Autoencoders for Dimensionality Reduction in Python**

Autoencoders can be implemented using **TensorFlow** and **Keras** in Python. The basic steps include:

1. **Loading the Dataset:** Common datasets such as **MNIST** or **CIFAR-10** are used to train autoencoders.
2. **Preprocessing the Data:** Normalization and reshaping of input images.
3. **Building the Autoencoder Model:**
   * A simple autoencoder consists of an **encoder (Dense layers)** followed by a **decoder (Dense layers)**.
4. **Training the Autoencoder:** Using an **Adam optimizer** and **Mean Squared Error (MSE) loss**.
5. **Evaluating the Performance:** Checking the quality of reconstructed images and analyzing dimensionality reduction.

**Advantages of Autoencoders for Dimensionality Reduction**

Compared to traditional techniques like PCA, autoencoders offer several benefits:

* **Non-linearity Handling:** Autoencoders can learn complex, non-linear relationships in data.
* **Better Representation Learning:** Captures hierarchical and meaningful features.
* **Scalability:** Works efficiently on large datasets without significant performance degradation.
* **Feature Learning without Labels:** Unlike supervised learning, autoencoders require no labeled data.

**Conclusion**

Autoencoders provide an efficient way to perform **dimensionality reduction**, especially for high-dimensional datasets. They are widely used in feature extraction, denoising, and anomaly detection tasks. With deep learning frameworks like **TensorFlow and Keras**, implementing autoencoders in Python has become straightforward. Their ability to learn efficient representations without supervision makes them a powerful tool in modern machine learning and deep learning applications.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, LeakyReLU

from tensorflow.keras.optimizers import RMSprop

from sklearn.manifold import TSNE

# Load Dataset

file\_path = "train.csv"  # Change if needed

df = pd.read\_csv(file\_path)

print(df.head())

# EDA

print(df.info())

print(df.describe())

print(df.isnull().sum().sum(), "missing values")

# Distribution of Target Variable

sns.histplot(df['target'], bins=30, kde=True)

plt.title("Target Variable Distribution")

plt.show()

# Remove ID and extract features

df.drop(columns=['ID\_code'], inplace=True)

X = df.drop(columns=['target']).values  # Features

y = df['target'].values  # Target

# Normalize Data

scaler = StandardScaler()

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

X\_train = scaler.fit\_transform(X\_train)

X\_test = scaler.transform(X\_test)

# Handle NaN or Inf values

X\_train = np.nan\_to\_num(X\_train)

X\_test = np.nan\_to\_num(X\_test)

# Autoencoder Architecture

encoding\_dim = 10  # Dimension reduction size

input\_dim = X\_train.shape[1]

input\_layer = Input(shape=(input\_dim,))

encoded = Dense(64)(input\_layer)

encoded = LeakyReLU(alpha=0.1)(encoded)

encoded = Dense(32)(encoded)

encoded = LeakyReLU(alpha=0.1)(encoded)

encoded = Dense(encoding\_dim, activation='linear')(encoded)

decoded = Dense(32)(encoded)

decoded = LeakyReLU(alpha=0.1)(decoded)

decoded = Dense(64)(decoded)

decoded = LeakyReLU(alpha=0.1)(decoded)

decoded = Dense(input\_dim, activation='linear')(decoded)

# Define Autoencoder

autoencoder = Model(input\_layer, decoded)

autoencoder.compile(optimizer=RMSprop(learning\_rate=0.0001), loss='mae')

# Train Autoencoder

autoencoder.fit(X\_train, X\_train, epochs=50, batch\_size=256, shuffle=True, validation\_data=(X\_test, X\_test))

# Extract Encoder

encoder = Model(input\_layer, encoded)

X\_train\_encoded = encoder.predict(X\_train)

X\_test\_encoded = encoder.predict(X\_test)

# t-SNE Visualization

tsne = TSNE(n\_components=2, random\_state=42)

X\_embedded = tsne.fit\_transform(X\_test\_encoded)

plt.scatter(X\_embedded[:, 0], X\_embedded[:, 1], c=y\_test[:len(X\_embedded)], cmap='coolwarm', alpha=0.5)

plt.colorbar()

plt.title("t-SNE Visualization of Encoded Features")

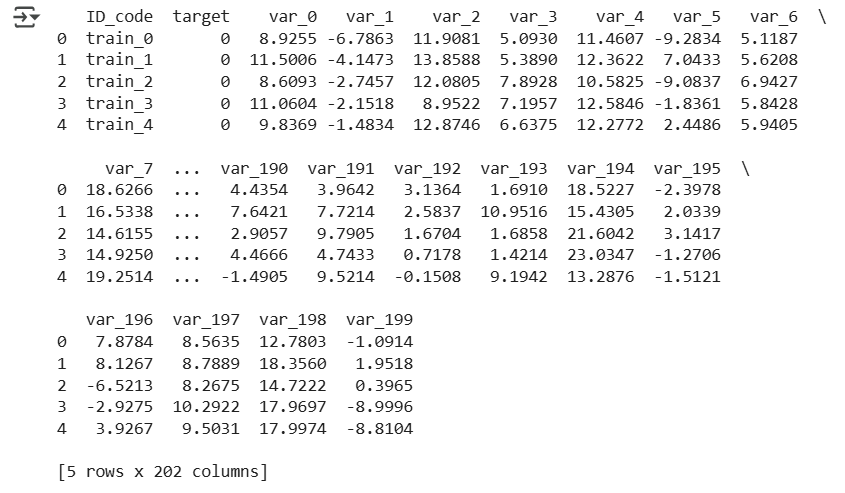
plt.show()

print("Original feature shape:", X\_train.shape)

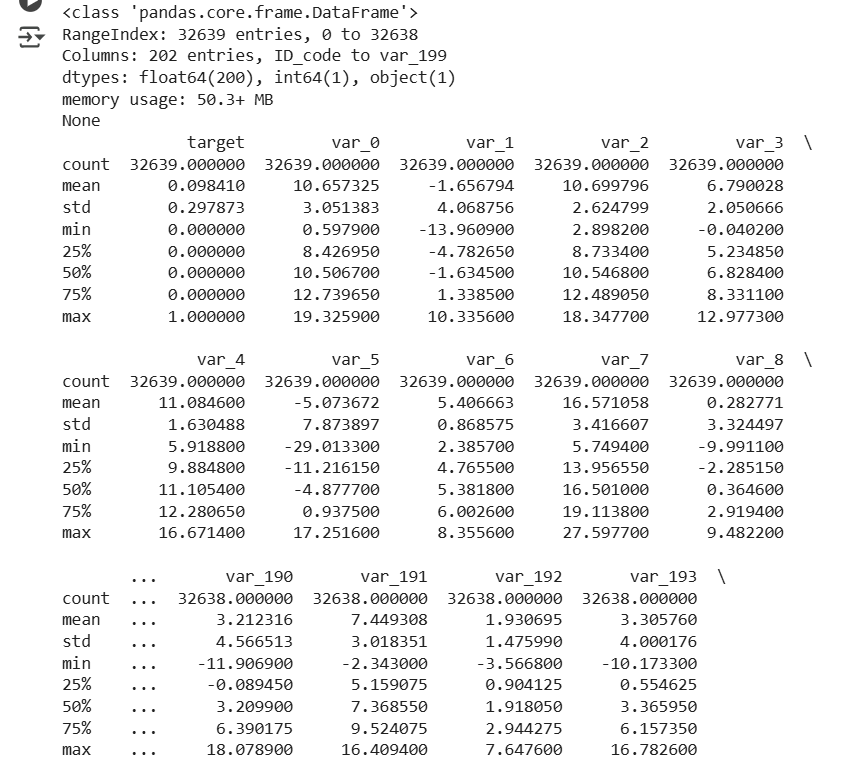
print("Reduced feature shape:", X\_train\_encoded.shape)

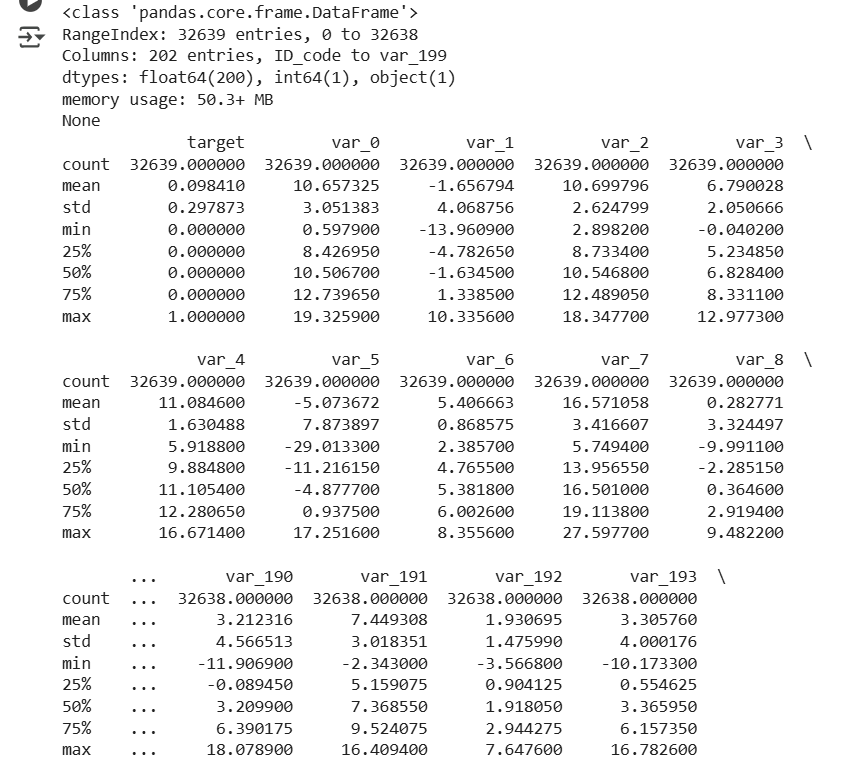
**Output:**

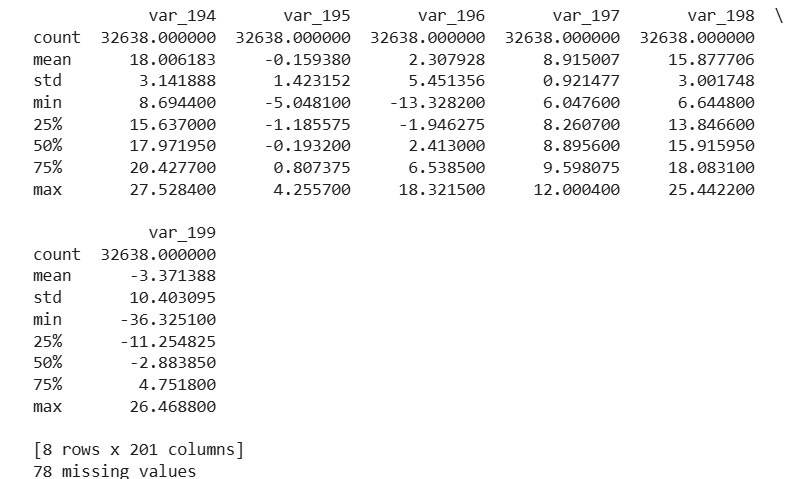
# Load dataset - df.head()



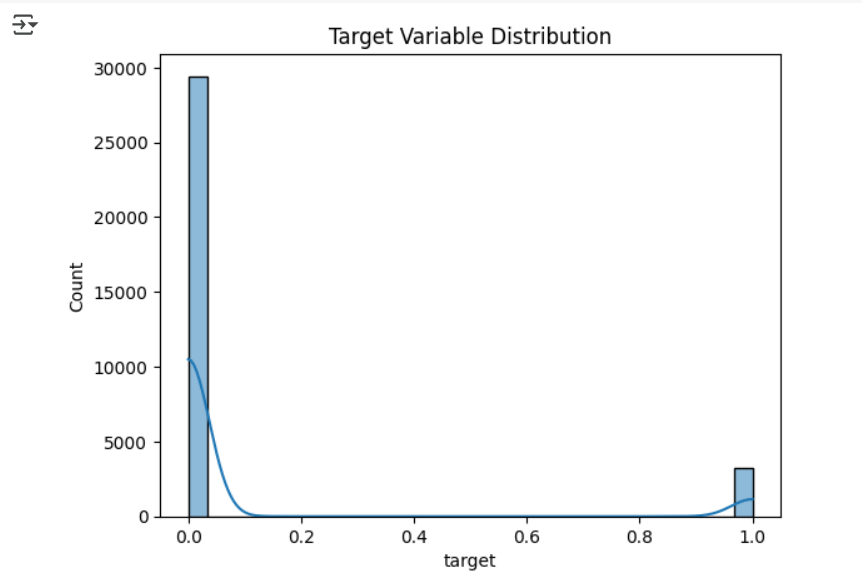
# EDA



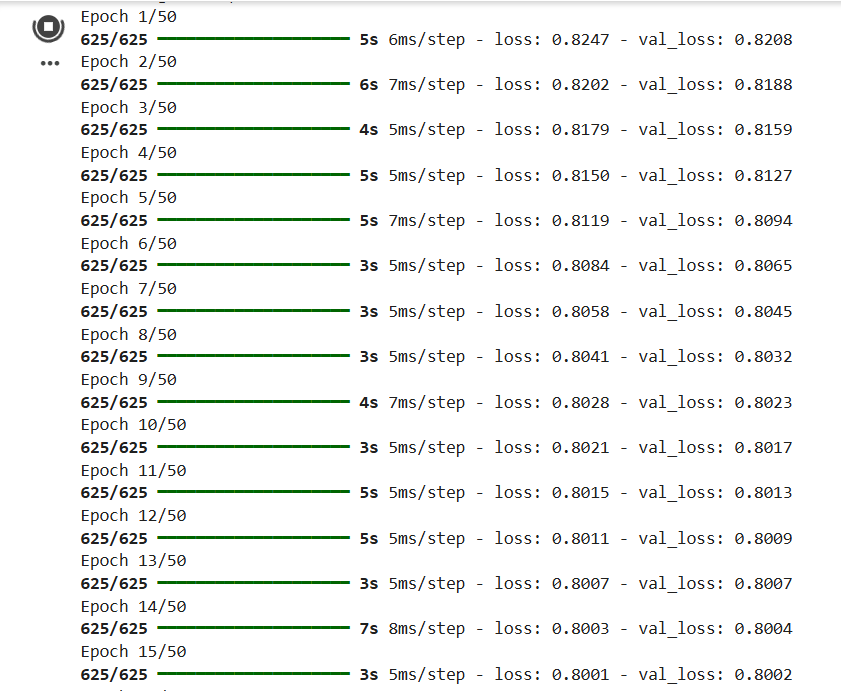




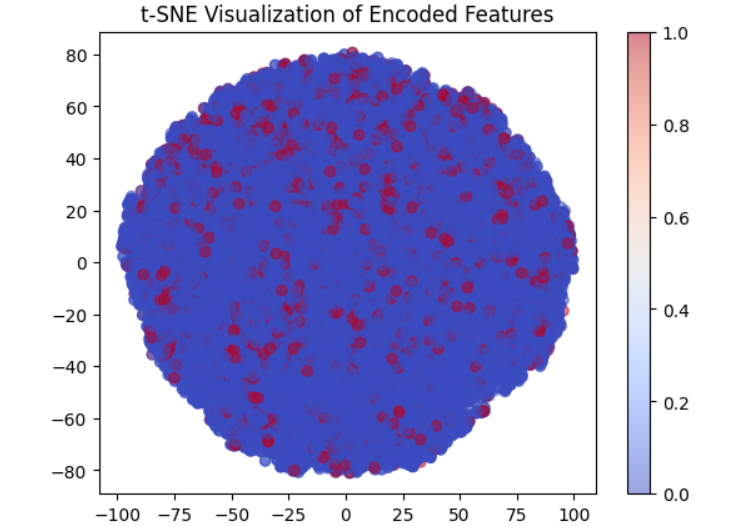
# Distribution of Target Variable



# autoencoder.fit()



# t-SNE Visualization



# Original vs Reduced



**Learning Outcomes:**

* Understand the concept of autoencoders and their role in dimensionality reduction.
* Learn to preprocess and normalize high-dimensional data for efficient training.

**Experiment 5**

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

**Objectives:**

* To implement Autoencoders in Python on Image Dataset.
* To train an Autoencoder using TensorFlow and Keras and analyze its reconstruction ability.

**Theory:**

**1. Introduction**

Autoencoders are a type of artificial neural network used for unsupervised learning of efficient codings in data. They are primarily utilized for dimensionality reduction, denoising, and feature extraction. In this experiment, we apply autoencoders to the MNIST dataset, a widely used benchmark dataset for handwritten digit recognition. We first implement a basic autoencoder using fully connected layers and later enhance its performance by employing convolutional layers.

**2. Autoencoders: Concept and Working Mechanism**

Autoencoders consist of two main components:

1. **Encoder**: This part compresses the input data into a lower-dimensional representation (latent space).
2. **Decoder**: This part reconstructs the input data from the compressed representation.

Mathematically, an autoencoder maps an input to a compressed representation using an encoding function , and then reconstructs using a decoding function.

The training objective is to minimize the reconstruction loss between and , usually measured by Mean Squared Error (MSE) or Binary Cross-Entropy (BCE).

**3. Application of Autoencoders on MNIST Dataset**

**3.1 Dataset Preparation**

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits (0-9), each of size 28x28 pixels. The data is normalized to the range to improve training efficiency.

**3.2 Fully Connected Autoencoder Implementation**

A simple autoencoder with fully connected layers is constructed as follows:

* **Encoder**: Input (784 dimensions) → Dense(128, ReLU) → Dense(64, ReLU) → Latent Space (32 dimensions)
* **Decoder**: Dense(64, ReLU) → Dense(128, ReLU) → Output (784 dimensions, Sigmoid)

After training with Adam optimizer and MSE loss function, the autoencoder is able to reconstruct the input images reasonably well. However, due to the dense architecture, spatial relationships in images are not efficiently captured.

**4. Improving Autoencoder Performance with Convolutional Layers**

To improve performance, a Convolutional Autoencoder (CAE) is implemented. Unlike fully connected layers, convolutional layers maintain spatial hierarchies, making them better suited for image reconstruction tasks.

**4.1 Convolutional Autoencoder Architecture**

* **Encoder**: Conv2D(32 filters, 3x3, ReLU) → MaxPooling2D(2x2) → Conv2D(64 filters, 3x3, ReLU) → MaxPooling2D(2x2)
* **Decoder**: Conv2DTranspose(64 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(32 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(1 filter, 3x3, Sigmoid)

**4.2 Performance Comparison**

The CAE significantly improves the quality of reconstructed images compared to the fully connected autoencoder. The main advantages include:

* **Preservation of spatial features**, reducing loss of information.
* **Faster convergence** due to fewer trainable parameters.
* **Better generalization** for unseen images.

**5. Conclusion**

Autoencoders provide a powerful method for image compression and feature extraction. While fully connected autoencoders serve as a baseline, convolutional autoencoders significantly enhance performance by leveraging spatial hierarchies. Future work can include Variational Autoencoders (VAEs) for more structured latent spaces and application of these techniques to real-world datasets beyond MNIST.

**Code:**

import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import pandas as pd

(X\_train, \_), (X\_test, \_) = tf.keras.datasets.fashion\_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(-1, 28 \* 28)

X\_test\_flat = X\_test.reshape(-1, 28 \* 28)

(X\_train, y\_train), (X\_test, y\_test) = tf.keras.datasets.fashion\_mnist.load\_data()

print(f"Train set shape: {X\_train.shape}, Labels: {y\_train.shape}")

print(f"Test set shape: {X\_test.shape}, Labels: {y\_test.shape}")

class\_labels = [

    "T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",

    "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"

]

plt.figure(figsize=(10, 6))

for i in range(10):

    plt.subplot(2, 5, i+1)

    plt.imshow(X\_train[i], cmap="gray")

    plt.title(class\_labels[y\_train[i]])

    plt.axis("off")

plt.tight\_layout()

plt.show()

class\_counts = pd.Series(y\_train).value\_counts().sort\_index()

plt.figure(figsize=(8, 5))

sns.barplot(x=class\_counts.index, y=class\_counts.values, palette="coolwarm")

plt.xticks(ticks=range(10), labels=class\_labels, rotation=45)

plt.xlabel("Class Labels")

plt.ylabel("Count")

plt.title("Distribution of Fashion MNIST Classes")

plt.show()

plt.figure(figsize=(8, 5))

sns.histplot(X\_train.flatten(), bins=50, kde=True, color="purple")

plt.xlabel("Pixel Intensity")

plt.ylabel("Frequency")

plt.title("Pixel Intensity Distribution in Fashion MNIST")

plt.show()

# Convert images to tuples for easy comparison

unique\_images = set([tuple(image.flatten()) for image in X\_train])

print(f"Total Images: {len(X\_train)}")

print(f"Unique Images: {len(unique\_images)}")

print(f"Duplicate Images: {len(X\_train) - len(unique\_images)}")

input\_dim = 28 \* 28  # 784 pixels

encoding\_dim = 64  # Reduced dimension

# Define the autoencoder

input\_layer = layers.Input(shape=(input\_dim,))

encoded = layers.Dense(128, activation='relu')(input\_layer)

encoded = layers.Dense(encoding\_dim, activation='relu')(encoded)

decoded = layers.Dense(128, activation='relu')(encoded)

decoded = layers.Dense(input\_dim, activation='sigmoid')(decoded)  # Output same shape as input

autoencoder = models.Model(input\_layer, decoded)

encoder = models.Model(input\_layer, encoded)  # Encoder for dimensionality reduction

autoencoder.compile(optimizer='adam', loss='mean\_squared\_error')

# Train the model

autoencoder.fit(X\_train\_flat, X\_train\_flat, epochs=20, batch\_size=256, validation\_data=(X\_test\_flat, X\_test\_flat))

# Reshape for CNN (add channel dimension)

X\_train = X\_train.reshape(-1, 28, 28, 1)

X\_test = X\_test.reshape(-1, 28, 28, 1)

# Convolutional Autoencoder

input\_layer = layers.Input(shape=(28, 28, 1))

# Encoder

x = layers.Conv2D(32, (3, 3), activation='relu', padding='same')(input\_layer)

x = layers.MaxPooling2D((2, 2), padding='same')(x)

x = layers.Conv2D(64, (3, 3), activation='relu', padding='same')(x)

x = layers.MaxPooling2D((2, 2), padding='same')(x)

encoded = layers.Conv2D(128, (3, 3), activation='relu', padding='same')(x)

# Decoder

x = layers.Conv2DTranspose(64, (3, 3), activation='relu', padding='same')(encoded)

x = layers.UpSampling2D((2, 2))(x)

x = layers.Conv2DTranspose(32, (3, 3), activation='relu', padding='same')(x)

x = layers.UpSampling2D((2, 2))(x)

decoded = layers.Conv2DTranspose(1, (3, 3), activation='sigmoid', padding='same')(x)

# Define model

autoencoder = models.Model(input\_layer, decoded)

# Compile and train

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

autoencoder.fit(X\_train, X\_train, epochs=20, batch\_size=256, validation\_data=(X\_test, X\_test))

encoder = models.Model(input\_layer, encoded)

X\_encoded = encoder.predict(X\_test)

print("Encoded shape:", X\_encoded.shape)

n = 10  # Number of images to display

decoded\_imgs = autoencoder.predict(X\_test)

plt.figure(figsize=(20, 4))

for i in range(n):

    # Original images

    plt.subplot(2, n, i + 1)

    plt.imshow(X\_test[i].reshape(28, 28), cmap='gray')

    plt.axis('off')

    # Reconstructed images

    plt.subplot(2, n, i + 1 + n)

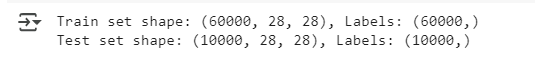
    plt.imshow(decoded\_imgs[i].reshape(28, 28), cmap='gray')

    plt.axis('off')

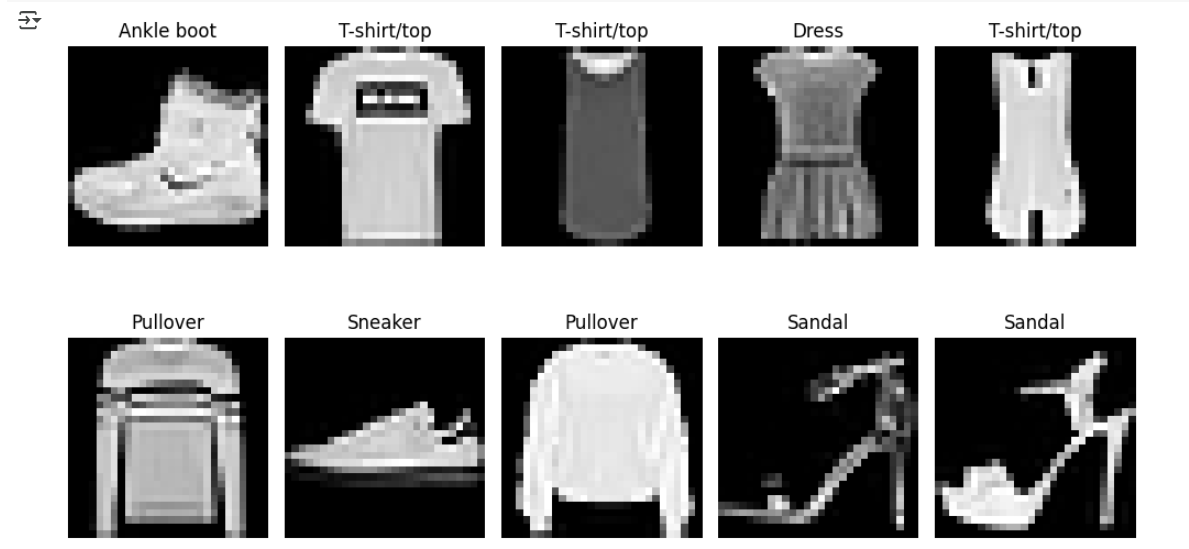
plt.show()

**Output:**

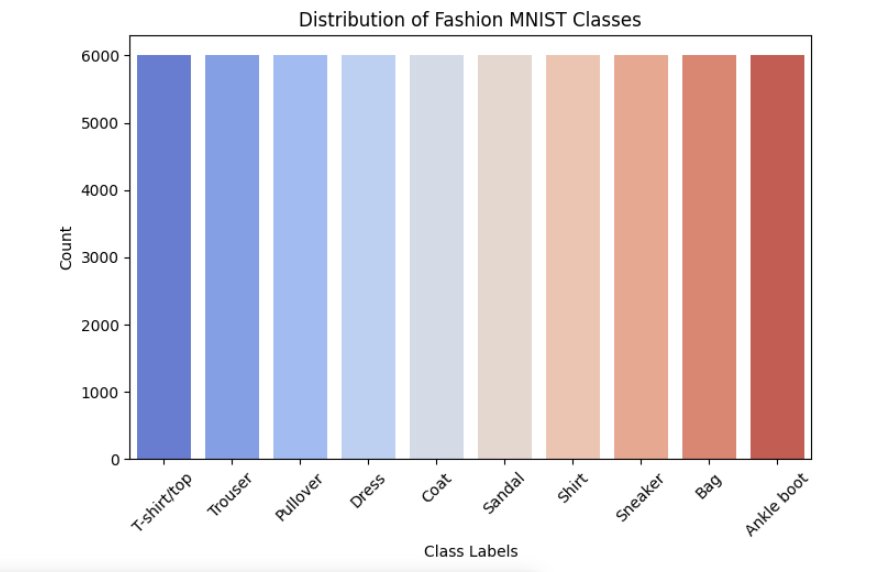
# Check dataset shape



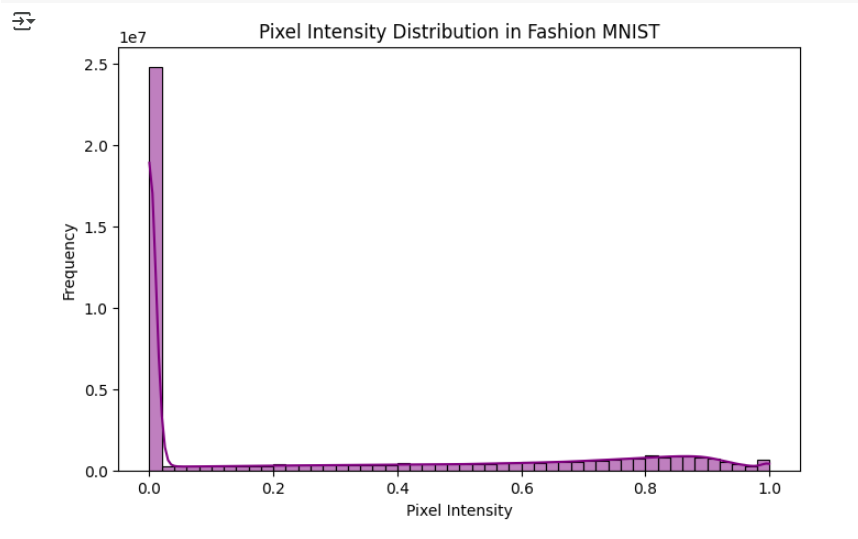
# Plot sample images with labels



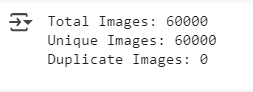
# Distribution of Fashion MNIST Classes



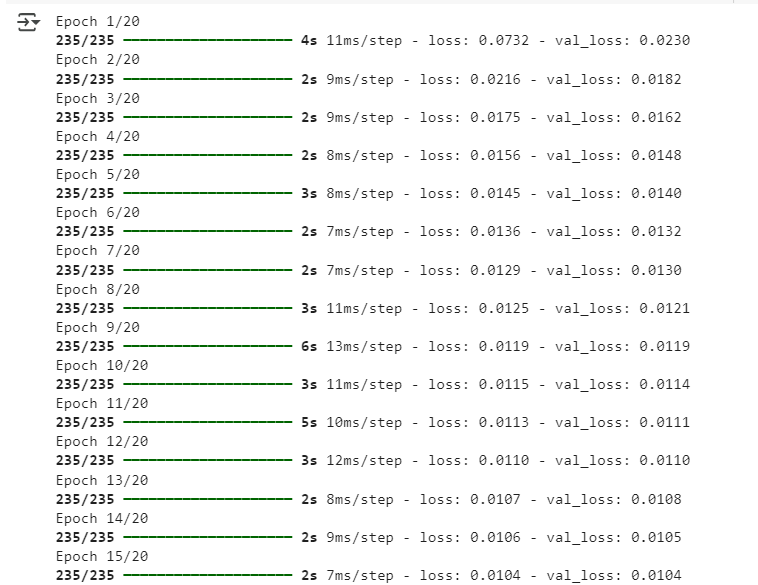
# Pixel Intensity Distribution in Fashion MNIST



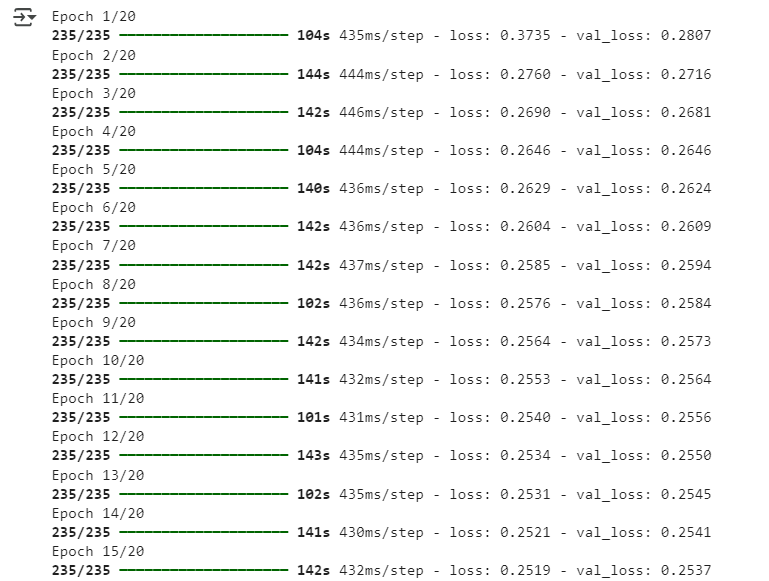
# Image count



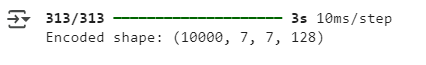
# Train the model



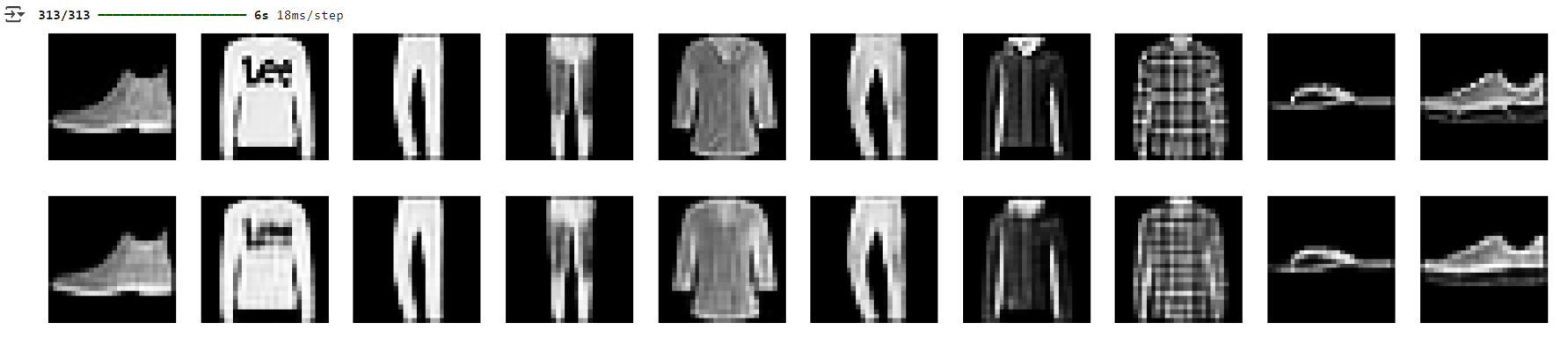
# Convolutional Autoencoder



# Encoded shape:



# Original vs Reconstructed



**Learning Outcomes:**

* Understand the concept of autoencoders and their application in image reconstruction.
* Learn to preprocess and normalize the MNIST dataset for improved training efficiency.

**Experiment 6**

**Aim: Application of Autoencoders on Image Dataset.**

**Theory:**

**1. Introduction**

Autoencoders are a type of artificial neural network used for unsupervised learning of efficient coding in data. They are primarily utilized for dimensionality reduction, denoising, and feature extraction. In this experiment, we apply autoencoders to the MNIST dataset, a widely used benchmark dataset for handwritten digit recognition. We first implement a basic autoencoder using fully connected layers and later enhance its performance by employing convolutional layers.

**2. Autoencoders: Concept and Working Mechanism**

Autoencoders consist of two main components:

1. **Encoder**: This part compresses the input data into a lower-dimensional representation (latent space).
2. **Decoder**: This part reconstructs the input data from the compressed representation.

Mathematically, an autoencoder maps an input to a compressed representation using an encoding function , and then reconstructs using a decoding function.

The training objective is to minimize the reconstruction loss between and , usually measured by Mean Squared Error (MSE) or Binary Cross-Entropy (BCE).

**3. Application of Autoencoders on MNIST Dataset**

**3.1 Dataset Preparation**

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits (0-9), each of size 28x28 pixels. The data is normalized to the range to improve training efficiency.

**3.2 Fully Connected Autoencoder Implementation**

A simple autoencoder with fully connected layers is constructed as follows:

* **Encoder**: Input (784 dimensions) → Dense(128, ReLU) → Dense(64, ReLU) → Latent Space (32 dimensions)
* **Decoder**: Dense(64, ReLU) → Dense(128, ReLU) → Output (784 dimensions, Sigmoid)

After training with Adam optimizer and MSE loss function, the autoencoder is able to reconstruct the input images reasonably well. However, due to the dense architecture, spatial relationships in images are not efficiently captured.

**4. Improving Autoencoder Performance with Convolutional Layers**

To improve performance, a Convolutional Autoencoder (CAE) is implemented. Unlike fully connected layers, convolutional layers maintain spatial hierarchies, making them better suited for image reconstruction tasks.

**4.1 Convolutional Autoencoder Architecture**

* **Encoder**: Conv2D(32 filters, 3x3, ReLU) → MaxPooling2D(2x2) → Conv2D(64 filters, 3x3, ReLU) → MaxPooling2D(2x2)
* **Decoder**: Conv2DTranspose(64 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(32 filters, 3x3, ReLU) → UpSampling2D(2x2) → Conv2DTranspose(1 filter, 3x3, Sigmoid)

**4.2 Performance Comparison**

The CAE significantly improves the quality of reconstructed images compared to the fully connected autoencoder. The main advantages include:

* **Preservation of spatial features**, reducing loss of information.
* **Faster convergence** due to fewer trainable parameters.
* **Better generalization** for unseen images.

**5. Conclusion**

Autoencoders provide a powerful method for image compression and feature extraction. While fully connected autoencoders serve as a baseline, convolutional autoencoders significantly enhance performance by leveraging spatial hierarchies. Future work can include Variational Autoencoders (VAEs) for more structured latent spaces and application of these techniques to real-world datasets beyond MNIST.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.layers import Input, Dense, Flatten, Reshape

from tensorflow.keras.models import Model

from tensorflow.keras.datasets import mnist

# Load MNIST dataset

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten the images

x\_train = x\_train.reshape((len(x\_train), 28\*28))

x\_test = x\_test.reshape((len(x\_test), 28\*28))

# Define the autoencoder architecture

encoding\_dim = 32  # Size of the encoded representation

# Encoder

input\_img = Input(shape=(28\*28,))

encoded = Dense(128, activation='relu')(input\_img)

encoded = Dense(64, activation='relu')(encoded)

encoded = Dense(encoding\_dim, activation='relu')(encoded)

# Decoder

decoded = Dense(64, activation='relu')(encoded)

decoded = Dense(128, activation='relu')(decoded)

decoded = Dense(28\*28, activation='sigmoid')(decoded)

autoencoder = Model(input\_img, decoded)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

# Train the autoencoder

autoencoder.fit(x\_train, x\_train, epochs=20, batch\_size=256, shuffle=True, validation\_data=(x\_test, x\_test))

# Evaluate on test images

encoded\_imgs = autoencoder.predict(x\_test)

encoded\_imgs = encoded\_imgs.reshape(-1, 28, 28)

# Display original and reconstructed images

def plot\_images(original, reconstructed, n=10):

    plt.figure(figsize=(20, 4))

    for i in range(n):

        # Original

        ax = plt.subplot(2, n, i + 1)

        plt.imshow(original[i], cmap='gray')

        plt.axis('off')

        # Reconstructed

        ax = plt.subplot(2, n, i + 1 + n)

        plt.imshow(reconstructed[i], cmap='gray')

        plt.axis('off')

    plt.show()

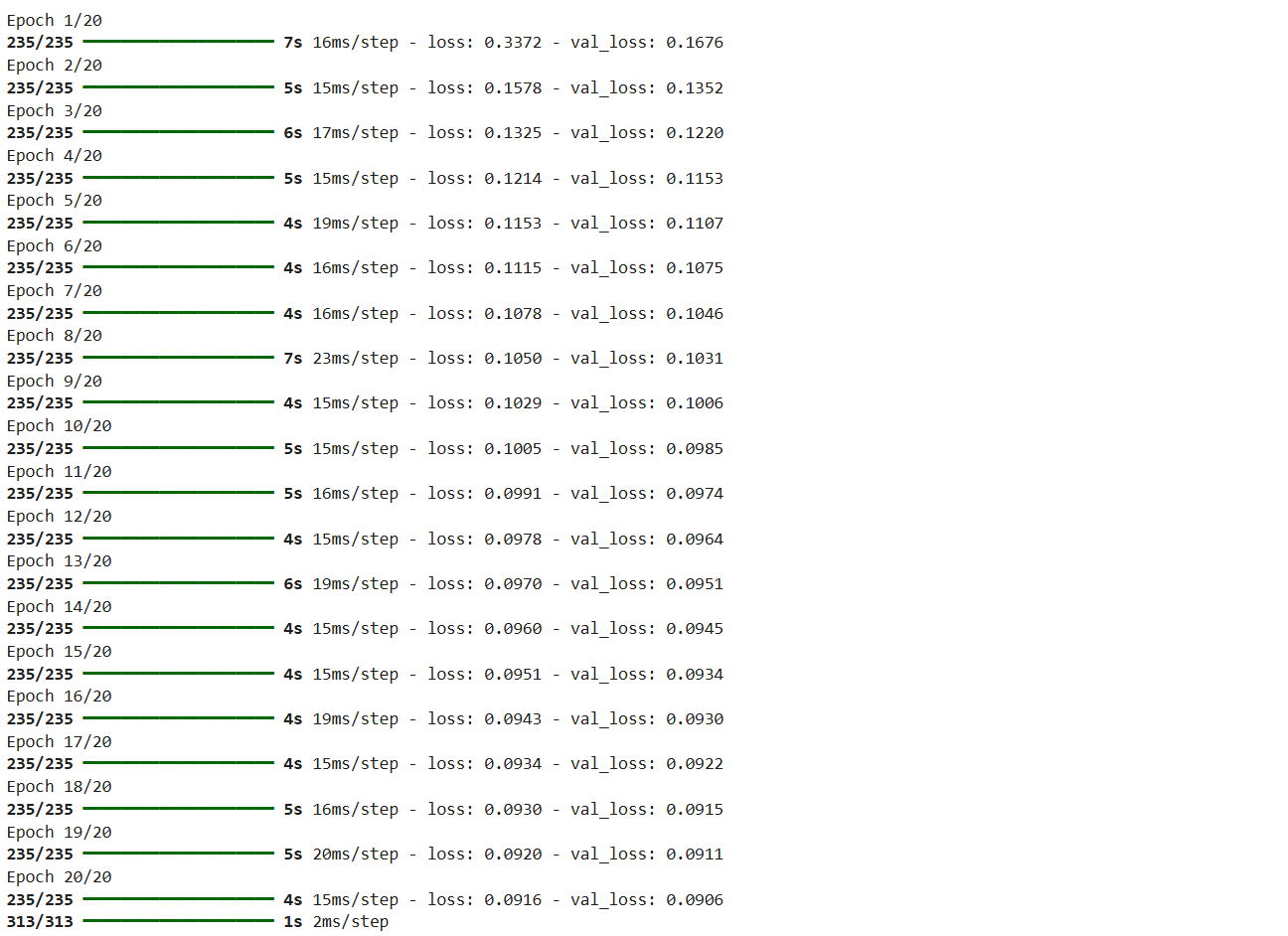
plot\_images(x\_test.reshape(-1, 28, 28), encoded\_imgs)

**Output:**

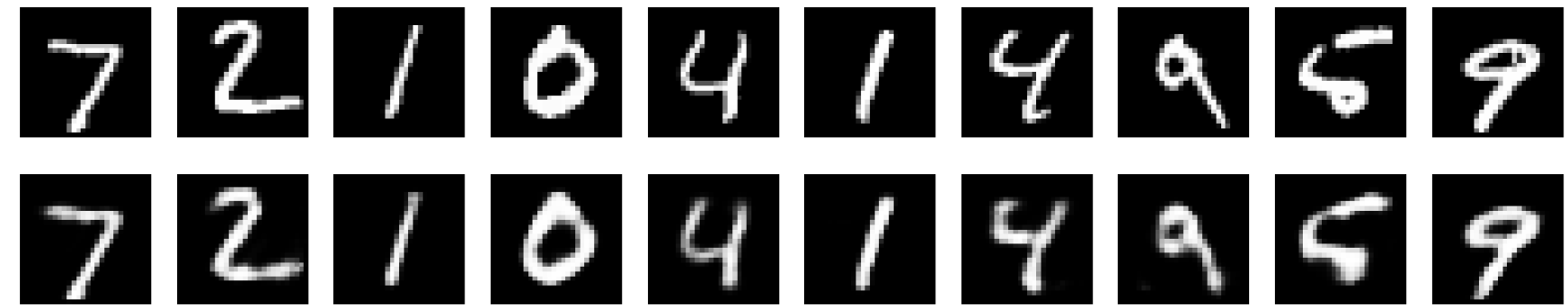
# Check dataset shape



# Training the model



# Final Output



**Learning Outcomes:**

* Understand the architecture and working of autoencoders.
* Implement autoencoders using TensorFlow/Keras for image datasets.
* Explore real-world applications like image denoising and anomaly detection.

**Experiment 7**

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

**Theory:**

**1. Introduction**

Stock price prediction is a crucial task in financial markets, where traders and investors aim to forecast future prices based on historical data. Recurrent Neural Networks (RNNs), specifically Long Short-Term Memory (LSTM) networks, are widely used for time series forecasting due to their ability to learn temporal dependencies. In this experiment, we implement an LSTM-based model for stock price prediction using historical stock data from Yahoo Finance. The model learns patterns from past prices and predicts future values based on these observations.

**2. Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM)**

**2.1 Concept and Working Mechanism**

Traditional RNNs are designed to process sequential data by maintaining a hidden state that captures information from previous time steps. However, they suffer from the vanishing gradient problem, making it difficult to learn long-term dependencies.

LSTM networks address this issue by introducing memory cells and gating mechanisms, which regulate the flow of information through the network:

* **Forget Gate:** Decides which information from the previous state should be discarded.
* **Input Gate:** Determines which new information should be stored in the cell state.
* **Output Gate:** Controls the output and updates the hidden state.

These mechanisms enable LSTMs to retain long-term dependencies, making them highly effective for time series forecasting.

**3. Application of LSTM in Stock Price Prediction**

**3.1 Dataset Preparation**

The dataset consists of historical stock prices, including Open, High, Low, Close, and Volume data, retrieved using the Yahoo Finance API. For training, we use only the 'Close' price, as it is a key indicator of stock movement. The data is preprocessed by:

* Normalizing prices using MinMax scaling to improve model convergence.
* Creating sequences of past prices to serve as input features for the LSTM model.
* Splitting the data into training and testing sets for model evaluation.

**3.2 LSTM Model Architecture**

The LSTM model is structured as follows:

* **Input Layer:** Takes sequences of past stock prices as input.
* **LSTM Layer:** Captures temporal dependencies and patterns in the stock price movements.
* **Dense Layer:** Outputs the predicted stock price.

Model architecture:

* LSTM(50 units, ReLU activation, return\_sequences=True)
* LSTM(50 units, ReLU activation)
* Dense(25 units, ReLU activation)
* Dense(1 unit, linear activation)

The model is trained using the Adam optimizer and Mean Squared Error (MSE) loss function.

**4. Performance Evaluation and Prediction**

After training, the model is evaluated on unseen test data. The predicted stock prices are compared with actual prices using:

* **Root Mean Squared Error (RMSE):** Measures prediction accuracy.
* **Visualization:** A graph comparing actual and predicted prices to assess performance.

While the model effectively captures trends, it may struggle with sudden market fluctuations caused by external factors (e.g., news events, economic policies).

**5. Conclusion**

LSTM networks provide a robust approach for stock price prediction by learning temporal dependencies in historical data. Despite their effectiveness, challenges such as market volatility and external influences limit their accuracy. Future improvements can include:

* Incorporating additional technical indicators (e.g., moving averages, RSI).
* Using hybrid models combining LSTMs with attention mechanisms.
* Exploring Transformer-based architectures for enhanced forecasting.

This experiment demonstrates the potential of deep learning in financial market analysis, paving the way for more sophisticated predictive models.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from sklearn.preprocessing import MinMaxScaler

import yfinance as yf

# Load stock data

def load\_stock\_data(stock\_symbol, start\_date, end\_date):

    df = yf.download(stock\_symbol, start=start\_date, end=end\_date)

    return df[['Close']]

# Prepare data

def prepare\_data(data, time\_step=60):

    scaler = MinMaxScaler(feature\_range=(0,1))

    data\_scaled = scaler.fit\_transform(data)

    X, y = [], []

    for i in range(time\_step, len(data\_scaled)):

        X.append(data\_scaled[i-time\_step:i, 0])

        y.append(data\_scaled[i, 0])

    return np.array(X), np.array(y), scaler

# Define LSTM model

def build\_lstm\_model(input\_shape):

    model = Sequential([

        LSTM(units=50, return\_sequences=True, input\_shape=input\_shape),

        Dropout(0.2),

        LSTM(units=50, return\_sequences=False),

        Dropout(0.2),

        Dense(units=25),

        Dense(units=1)

    ])

    model.compile(optimizer='adam', loss='mean\_squared\_error')

    return model

# Load data

data = load\_stock\_data('AAPL', '2020-01-01', '2023-01-01')

# Prepare dataset

time\_step = 60

X, y, scaler = prepare\_data(data.values, time\_step)

X = X.reshape(X.shape[0], X.shape[1], 1)  # Reshape for LSTM

# Split data

train\_size = int(len(X) \* 0.8)

X\_train, y\_train = X[:train\_size], y[:train\_size]

X\_test, y\_test = X[train\_size:], y[train\_size:]

# Build and train model

model = build\_lstm\_model((X\_train.shape[1], 1))

model.fit(X\_train, y\_train, epochs=50, batch\_size=32, validation\_data=(X\_test, y\_test))

# Make predictions

y\_pred = model.predict(X\_test)

y\_pred = scaler.inverse\_transform(y\_pred.reshape(-1,1))

y\_test = scaler.inverse\_transform(y\_test.reshape(-1,1))

# Plot results

plt.figure(figsize=(12,6))

plt.plot(y\_test, label='Actual Prices')

plt.plot(y\_pred, label='Predicted Prices')

plt.legend()

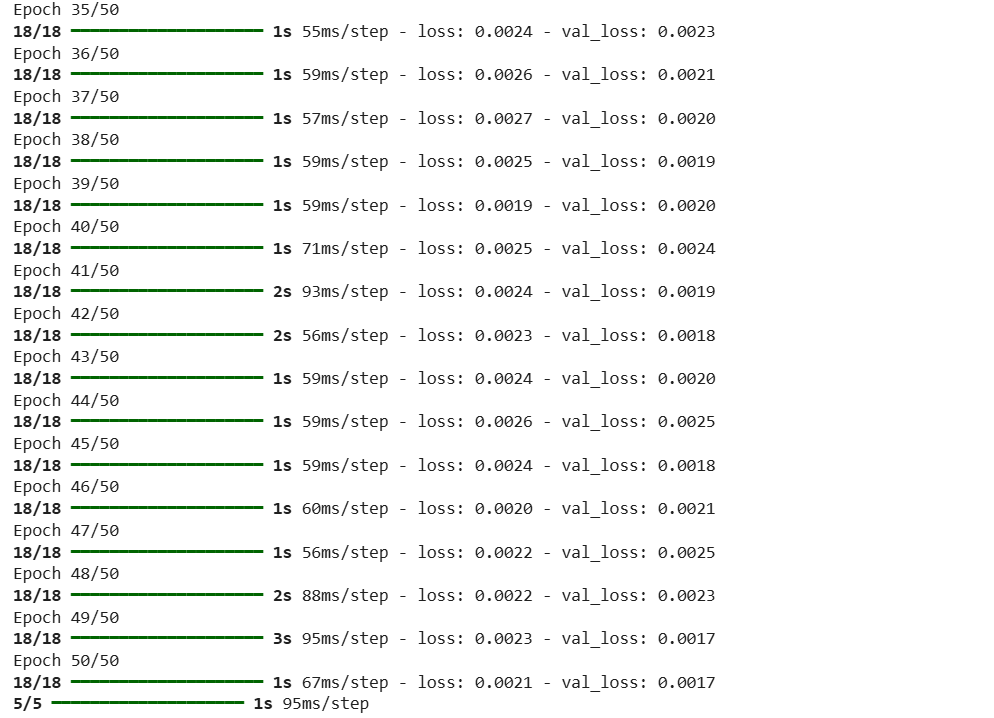
plt.show()

**Output:**

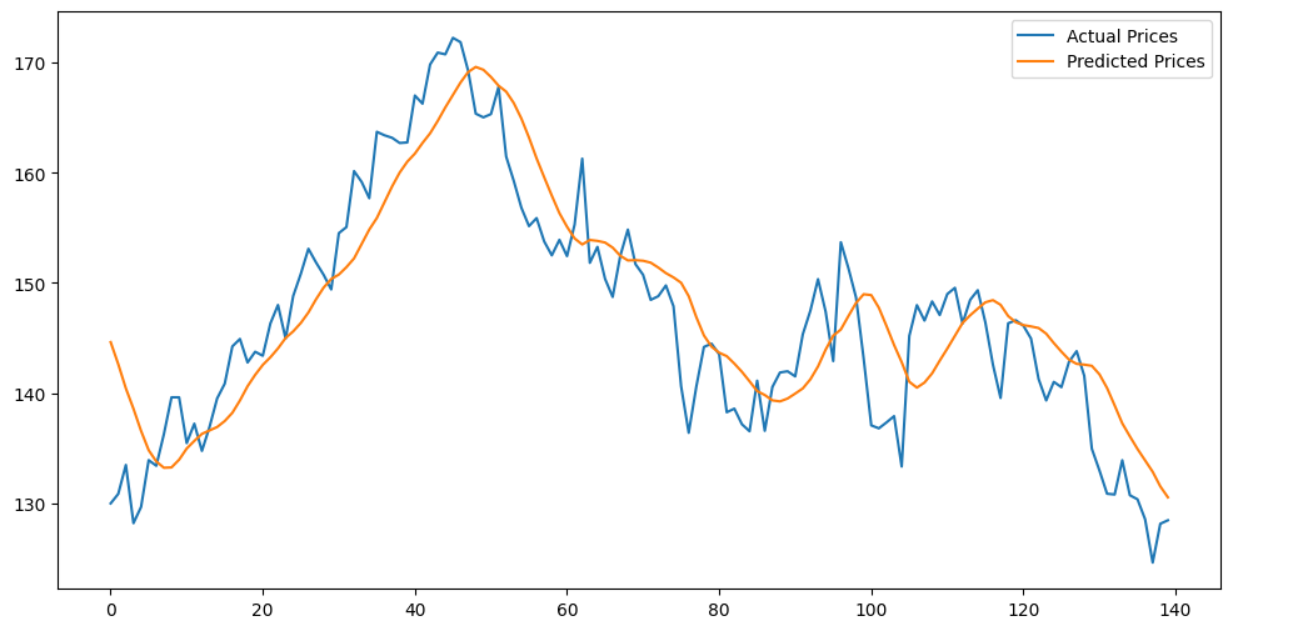
# Load Data



# Training the model



# Final Output : Plot results



**Learning Outcomes:**

* **Applied LSTMs for time series forecasting** to capture stock price trends.
* **Learned data preprocessing techniques** like normalization and sequence creation.
* **Evaluated model performance** and understood challenges in stock prediction.

**Experiment 8**

**Aim: Implementation of transfer learning using the pre-trained model (VGG16) on image dataset in Python.**

**Theory:**

**Transfer learning using pre-trained VGG16 for image classification**

Transfer learning enables the reuse of knowledge gained by a model trained on a large dataset for a different but related task. In image classification, pre-trained models like VGG16, trained on ImageNet with over a million images across 1000 categories, serve as strong foundations for extracting visual features from new datasets. This experiment aims to implement transfer learning using the VGG16 model and evaluate the improvements achieved by leveraging its pre-learned weights instead of training from scratch.

**VGG16 Architecture and Role in Transfer Learning**

VGG16 is a deep convolutional neural network with 16 weight layers, known for its uniform architecture and strong performance in image classification tasks. It uses small 3x3 convolution filters, arranged in increasing depth, making the network both deep and efficient at capturing hierarchical image features.

**Main components of VGG16 include:**

* **Input Layer** – Accepts images of size 224x224x3.
* **Convolutional Layers (13 total)** – Use 3x3 filters with stride 1 and ‘same’ padding; ReLU is applied after each convolution to introduce non-linearity.
* **Max Pooling Layers** – 2x2 max pooling applied after every few convolution layers to reduce spatial dimensions.
* **Fully Connected Layers** – Two dense layers with 4096 neurons each and ReLU activation, followed by a final dense layer with 1000 outputs (for ImageNet) and softmax activation.
* **Dropout Layers** – Applied between fully connected layers to prevent overfitting.
* **Output Layer** – Softmax activation used for classification.

By using the convolutional base of VGG16 as a fixed feature extractor or fine-tuning some of its layers, new image classification models can quickly generalize to novel tasks without needing massive datasets or training time.

**Process of Image Classification using Transfer Learning**

The classification pipeline involves importing the pre-trained VGG16 model (without the top classifier), attaching custom layers, and training only these new layers while keeping the rest frozen. Alternatively, a few deeper layers of VGG16 can be unfrozen for fine-tuning.

1. **Data Preprocessing -**

Input images are resized (typically to 224x224), normalized, and augmented to increase dataset variability. Data generators are used to handle image loading and transformation in real-time during training.

1. **Model Construction -**

The VGG16 model is loaded with include\_top=False and pre-trained ImageNet weights. On top of the frozen convolutional base, custom layers like GlobalAveragePooling2D, Dense, Dropout, and a final softmax Dense layer are added.

1. **Training the Model -**

The model is compiled using optimizers like Adam and the categorical crossentropy loss function for multi-class classification. Training is done in batches using the training set, while performance is monitored on the validation set.

1. **Evaluation and Performance Measurement -**

Once trained, the model is tested on unseen data. The benefit of using VGG16 lies in its ability to quickly generalize using fewer data points. The trained model’s accuracy, precision, recall, and F1-score are evaluated and compared to a baseline model trained from scratch.

**Model Evaluation and Visualization**

Evaluating transfer learning involves observing how well the model generalizes to the target task. Confusion matrices, classification reports, and accuracy/loss curves are plotted to analyze learning behavior.

Performance Metrics:

* **Accuracy** – Measures overall correctness of predictions.
* **Precision & Recall** – Assess model reliability per class.
* **F1-Score** – Harmonic mean of precision and recall.
* **Training Time** – Shorter with transfer learning compared to full training.

**Visualization of Results**

Visual tools like training/validation accuracy plots and confusion matrices help identify overfitting, misclassifications, and class-wise performance. These insights reveal how efficiently the VGG16 features are reused and adapted.

**Conclusion**

Transfer learning with VGG16 significantly reduces training time and enhances model performance on small or domain-specific datasets. By utilizing pre-learned visual patterns, the model requires fewer resources while achieving high accuracy. The evaluation confirms that feature reuse through VGG16 is a powerful strategy in image classification tasks.

**Code:**

import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import classification\_report, confusion\_matrix

import seaborn as sns

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.applications import VGG16

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten, Dropout

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.applications.vgg16 import preprocess\_input

# Load and preprocess the CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

x\_train = preprocess\_input(x\_train.astype('float32'))

x\_test = preprocess\_input(x\_test.astype('float32'))

y\_train\_cat = to\_categorical(y\_train, 10)

y\_test\_cat = to\_categorical(y\_test, 10)

# Load pre-trained VGG16 without the top layer

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(32, 32, 3))

# Freeze base layers

for layer in base\_model.layers:

    layer.trainable = False

# Add custom top layers

x = Flatten()(base\_model.output)

x = Dense(256, activation='relu')(x)

x = Dropout(0.5)(x)

x = Dense(10, activation='softmax')(x)

# Final model

model = Model(inputs=base\_model.input, outputs=x)

model.compile(optimizer=Adam(), loss='categorical\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(x\_train, y\_train\_cat, epochs=10, batch\_size=64, validation\_data=(x\_test, y\_test\_cat))

# Evaluate the model

loss, accuracy = model.evaluate(x\_test, y\_test\_cat)

print(f"Test Accuracy: {accuracy:.4f}")

# Predict and generate classification report

y\_pred = model.predict(x\_test)

y\_pred\_classes = np.argmax(y\_pred, axis=1)

y\_true = y\_test.flatten()

print("\nClassification Report:")

print(classification\_report(y\_true, y\_pred\_classes))

# Confusion Matrix

cm = confusion\_matrix(y\_true, y\_pred\_classes)

plt.figure(figsize=(10,7))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')

plt.title("Confusion Matrix")

plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.show()

# Plot accuracy and loss

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

plt.subplot(1,2,1)

plt.plot(history.history['accuracy'], label='Train Acc')

plt.plot(history.history['val\_accuracy'], label='Val Acc')

plt.title("Model Accuracy")

plt.xlabel("Epochs")

plt.ylabel("Accuracy")

plt.legend()

plt.subplot(1,2,2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Val Loss')

plt.title("Model Loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.tight\_layout()

plt.show()

**Output:**

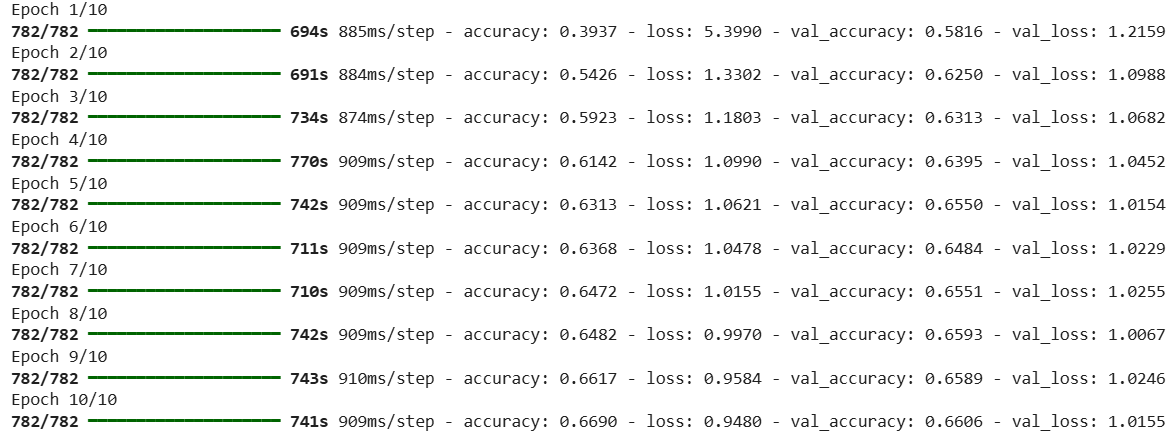
# Load and preprocess the CIFAR-10 dataset



# Load pre-trained VGG16 without the top layer



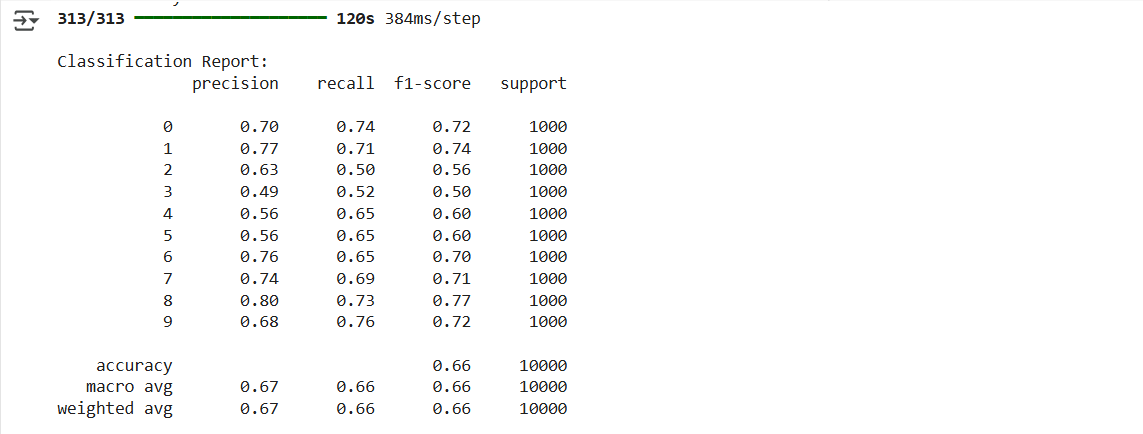
# Train the model



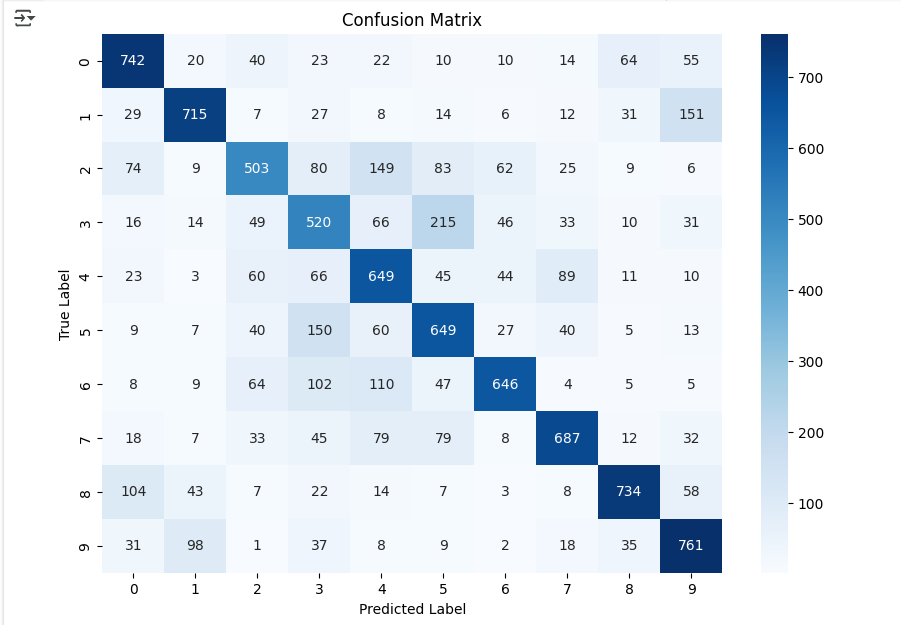
# Evaluate the model



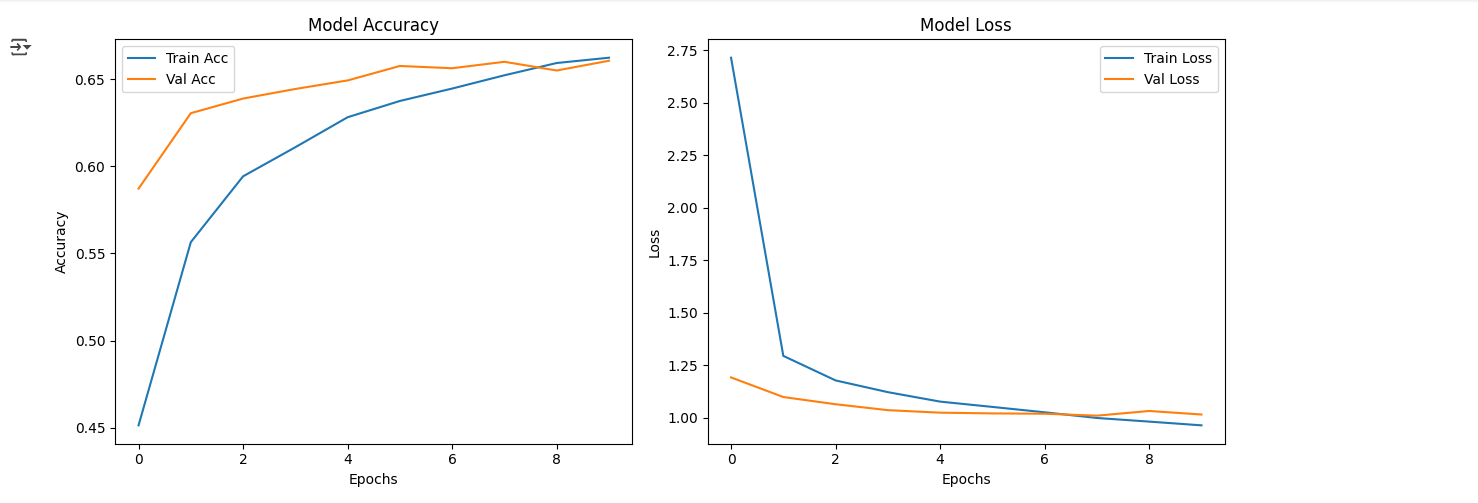
# Classification report



# Confusion Matrix

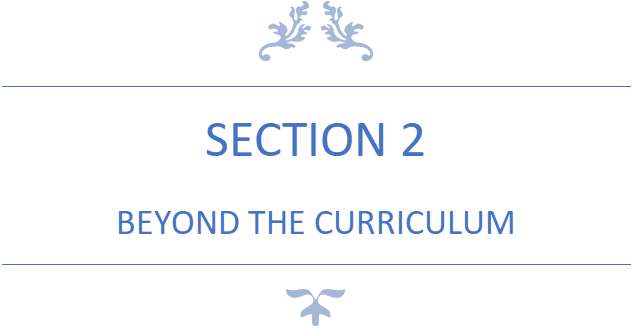


# Plot accuracy and loss

****

**Learning Outcomes:**

1. Understood the application of transfer learning using the pre-trained VGG16 model for efficient image classification.
2. Learned the structure and functionality of VGG16, including its convolutional layers, pooling layers, and fully connected layers.
3. Gained the ability to evaluate and interpret model performance using classification metrics and result visualizations.



**EXPERIMENT-1**

**Aim: Implement an Inception V3 for image classification.**

**Objectives:**

* To implement the Inception V3 model for efficient and accurate image classification.
* To evaluate the performance benefits of using a deep, multi-scale convolutional architecture.

**Theory:**

**Image Classification using Inception V3**

The Inception V3 model is a state-of-the-art convolutional neural network architecture known for its optimized depth, width, and utilization of computational resources. It is specifically designed for large-scale image classification tasks where extracting intricate patterns and multi-scale features is crucial. The primary goal of this experiment is to implement the Inception V3 architecture for accurate image classification while analyzing the improvements gained by its multi-branch, multi-resolution design. This deep model efficiently balances performance and speed, making it suitable for both academic and industrial image classification tasks.

**Inception V3 Architecture and Functionality**

Inception V3 improves upon earlier versions of the Inception model by integrating advanced concepts like factorized convolutions, label smoothing, batch normalization, and auxiliary classifiers. These enhancements collectively help in accelerating convergence, reducing overfitting, and improving generalization. The architecture is built using a stack of Inception modules, where each module performs multiple convolution operations in parallel using filters of different sizes. This allows the model to learn spatial hierarchies of features efficiently.

**Main components of Inception V3:**

* **Stem Convolutional Layers** – Initial layers that extract low-level features from raw input images.
* **Inception Modules** – Core blocks that apply multiple types of convolutions and pooling in parallel to capture diverse features.
* **Factorized Convolutions** – Replace large filters (e.g., 5x5) with smaller ones (e.g., two 3x3s) to reduce the number of parameters.
* **Auxiliary Classifiers** – Intermediate output branches used during training to ensure better gradient flow and reduce vanishing gradients.
* **Batch Normalization** – Applied after most layers to speed up training and provide regularization.
* **Global Average Pooling** – Compresses spatial dimensions by averaging feature maps, reducing overfitting.
* **Dropout and Dense Layers** – Dropout helps in regularization, and dense layers map learned features to output class probabilities.

**Process of Image Classification using Inception V3**

1. **Data Preprocessing -**

Input images are resized to 299x299 pixels, normalized, and optionally augmented using techniques such as horizontal flips, rotations, and zooms. This step enhances the model’s robustness to real-world image variations and improves generalization.

1. **Model Construction -**

Inception V3 can be used in two ways: (a) from scratch with randomly initialized weights, or (b) by leveraging a pre-trained version on ImageNet and fine-tuning it for a specific dataset. The top classification layer is modified to match the number of classes in the current task. The model is compiled using categorical crossentropy for multi-class classification and an appropriate optimizer like RMSprop or Adam.

1. **Training the Model -**

The model is trained on training data while validating on a separate validation set. During each epoch, weights are adjusted to minimize the classification error. Callbacks like early stopping and learning rate scheduling may be used to stabilize and optimize training. Auxiliary classifiers assist in faster convergence and deeper learning by injecting gradients into earlier layers.

1. **Evaluation and Prediction -**

Once trained, the model is evaluated on a test set using performance metrics. The predicted labels are compared with actual labels to assess classification accuracy. The final output is a probability distribution over classes, from which the highest-probability class is chosen as the predicted label.

**Model Evaluation and Visualization**

Evaluation helps determine how effectively the model classifies unseen images and generalizes to new data.

Performance Metrics:

* **Accuracy** – Indicates overall correctness of predictions.
* **Precision and Recall** – Assess classification performance for each class.
* **F1-Score** – Balances precision and recall for better evaluation in imbalanced datasets.
* **Confusion Matrix** – Helps visualize true positives, false positives, and misclassifications across classes.

**Visualization of Results:**

Plotting training vs. validation loss and accuracy over epochs gives insight into overfitting or underfitting. Class activation maps (CAMs) or Grad-CAMs may also be used to visualize which parts of the image influenced the model's predictions, enhancing interpretability and trust in the model.

**Conclusion**

The Inception V3 model is an advanced deep learning architecture that brings significant improvements to image classification through its scalable and efficient design. It effectively captures both local and global patterns within images by combining convolutions of different sizes. Using techniques like factorized convolutions, auxiliary classifiers, and batch normalization, the model achieves a strong balance between performance and computational efficiency. Evaluation with metrics and visualizations confirms its ability to generalize well, making it highly suitable for complex image classification tasks in real-world scenarios.

**Code and Outputs:**

import os

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.preprocessing.image import ImageDataGenerator

from tensorflow.keras.applications.inception\_v3 import InceptionV3, preprocess\_input

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, GlobalAveragePooling2D

from tensorflow.keras.optimizers import Adam

from sklearn.metrics import classification\_report, confusion\_matrix

import seaborn as sns

train\_dir = '/content/sports\_dataset/sports\_dataset/train'

test\_dir = '/content/sports\_dataset/sports\_dataset/test'

train\_datagen = ImageDataGenerator(

    preprocessing\_function=preprocess\_input,

    rotation\_range=20,

    zoom\_range=0.2,

    horizontal\_flip=True

)

test\_datagen = ImageDataGenerator(preprocessing\_function=preprocess\_input)

train\_generator = train\_datagen.flow\_from\_directory(

    train\_dir,

    target\_size=(299, 299),

    batch\_size=32,

    class\_mode='categorical'

)

test\_generator = test\_datagen.flow\_from\_directory(

    test\_dir,

    target\_size=(299, 299),

    batch\_size=32,

    class\_mode='categorical',

    shuffle=False

)

# Load base model

base\_model = InceptionV3(weights='imagenet', include\_top=False)

x = base\_model.output

x = GlobalAveragePooling2D()(x)

x = Dense(1024, activation='relu')(x)

predictions = Dense(train\_generator.num\_classes, activation='softmax')(x)

model = Model(inputs=base\_model.input, outputs=predictions)

# Freeze base layers

for layer in base\_model.layers:

    layer.trainable = False

# Compile model

model.compile(optimizer=Adam(learning\_rate=0.0001),

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

# Train model

history = model.fit(

    train\_generator,

    epochs=10,

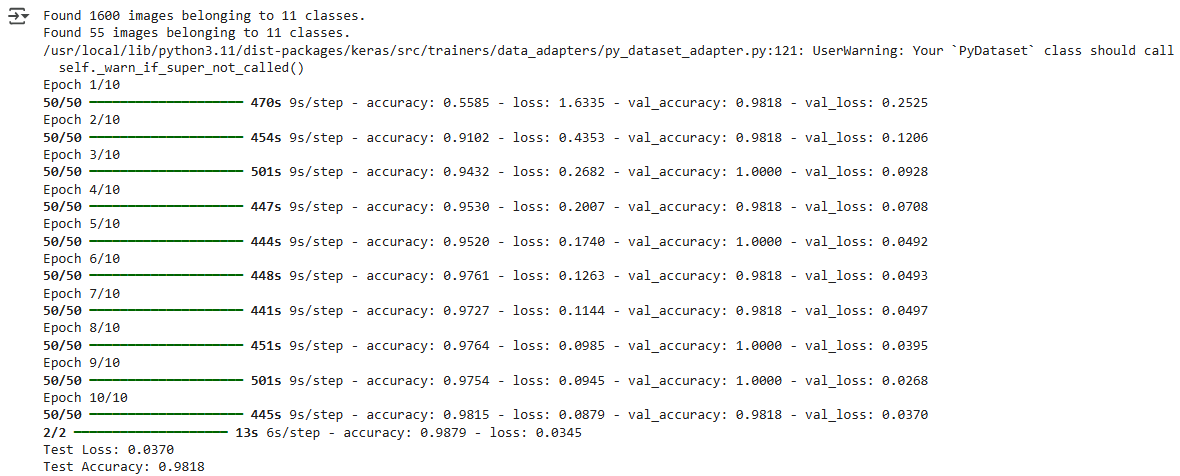
    validation\_data=test\_generator

)

loss, accuracy = model.evaluate(test\_generator)

print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")



predictions = model.predict(test\_generator)

y\_pred = np.argmax(predictions, axis=1)

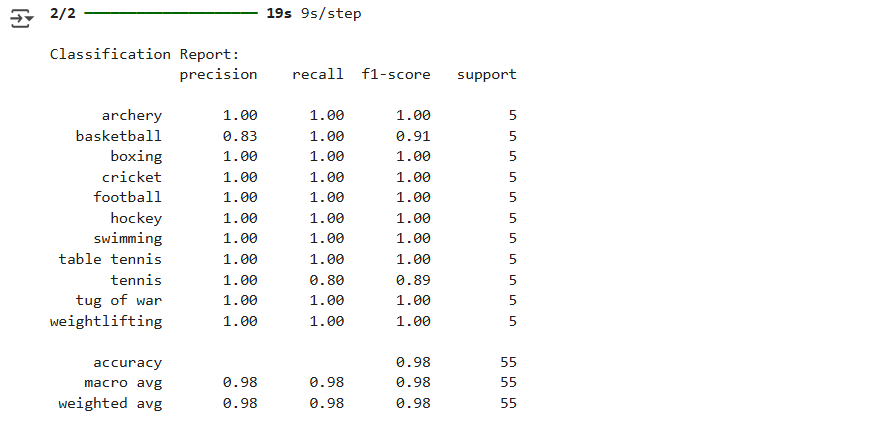
y\_true = test\_generator.classes

class\_labels = list(test\_generator.class\_indices.keys())

# Classification report

print("\nClassification Report:")

print(classification\_report(y\_true, y\_pred, target\_names=test\_generator.class\_indices.keys()))



def predict\_image(img\_path, model, class\_labels):

    img = image.load\_img(img\_path, target\_size=(299, 299))

    x = image.img\_to\_array(img)

    x = np.expand\_dims(x, axis=0)

    x = preprocess\_input(x)

    preds = model.predict(x)

    predicted\_class = class\_labels[np.argmax(preds)]

    return predicted\_class

image\_paths\_and\_labels = []

for class\_name in class\_labels:

    folder\_path = os.path.join(test\_dir, class\_name)

    for filename in os.listdir(folder\_path):

        if filename.endswith(('.jpg', '.jpeg', '.png')):

            image\_path = os.path.join(folder\_path, filename)

            image\_paths\_and\_labels.append((image\_path, class\_name))

num\_images\_to\_display = 9

random\_images = random.sample(image\_paths\_and\_labels, num\_images\_to\_display)

plt.figure(figsize=(12, 12))

for i, (img\_path, true\_label) in enumerate(random\_images):

    predicted\_class = predict\_image(img\_path, model, class\_labels)

    plt.subplot(3, 3, i + 1)

    img = image.load\_img(img\_path, target\_size=(299, 299))

    plt.imshow(img)

    plt.title(f"Pred: {predicted\_class}\nTrue: {true\_label}")

    plt.axis('off')

plt.tight\_layout()

plt.show()



# Confusion matrix

cm = confusion\_matrix(y\_true, y\_pred)

plt.figure(figsize=(12,10))

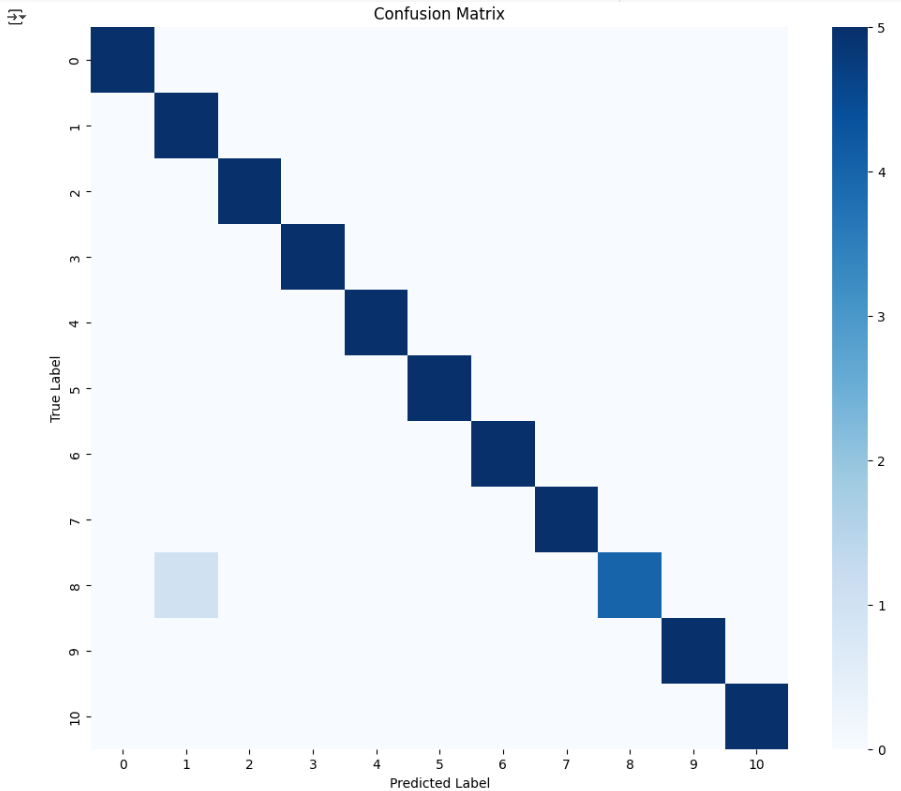
sns.heatmap(cm, annot=False, fmt='d', cmap='Blues')

plt.title('Confusion Matrix')

plt.ylabel('True Label')

plt.xlabel('Predicted Label')

plt.show()



plt.figure(figsize=(10,4))

plt.subplot(1,2,1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Val Accuracy')

plt.title('Accuracy Over Epochs')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.subplot(1,2,2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Val Loss')

plt.title('Loss Over Epochs')

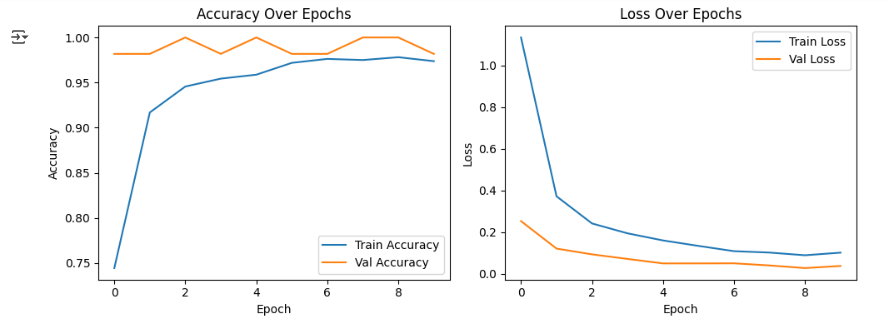
plt.xlabel('Epoch')

plt.ylabel('Loss')

plt.legend()

plt.tight\_layout()

plt.show()

****

**Learning Outcomes:**

1. Gained understanding of Inception V3 architecture and its multi-scale convolutional approach for efficient image classification.
2. Developed skills to preprocess, train, and fine-tune a deep learning model using real-world image datasets.
3. Acquired the ability to evaluate model performance using metrics like accuracy, F1-score, and visualize results for deeper insights.

**EXPERIMENT-2**

**Aim: Implement Bi-directional LSTM for text classification.**

**Objectives:**

* To implement a Bi-directional LSTM model for accurate text classification.
* To analyze the impact of processing text sequences in both forward and backward directions.

**Theory:**

**Text classification using Bi-directional LSTM**

Bi-directional Long Short-Term Memory (Bi-LSTM) networks are an advanced deep learning architecture used in Natural Language Processing (NLP), specifically designed to improve the performance of sequence-based tasks like text classification. This experiment aims to implement a Bi-LSTM model to accurately classify text data while analyzing the advantages of processing input sequences in both forward and backward directions. By capturing context from both ends of a sentence, Bi-LSTM models help overcome the limitations of standard LSTM, which can only learn from past information.

**Bi-LSTM Architecture and Functionality**

In Bi-LSTM, two separate LSTM layers are employed: one reads the input sequence from left to right (forward pass), and the other reads from right to left (backward pass). The outputs of both directions are concatenated at each time step, allowing the model to understand both preceding and succeeding words. This is crucial for tasks where word meaning heavily depends on surrounding context. For example, in the sentence "He **banked** the fire," the meaning of *banked* changes if we don’t consider the full context.

**Main components of the Bi-LSTM model include:**

* **Embedding Layer** – Transforms input words into dense vector representations that carry semantic meaning. Pre-trained embeddings like Word2Vec or GloVe can enhance this further.
* **Bi-LSTM Layer** – Comprises two LSTM layers operating in opposite directions; they extract both past and future dependencies for deeper context understanding.
* **Dropout Layer** – Randomly drops units to reduce overfitting and improve model generalization.
* **Dense Layer** – Processes the combined outputs of the Bi-LSTM and projects them to the desired output dimension.
* **Output Layer** – Applies softmax or sigmoid activation to produce class probabilities based on binary or multi-class classification setup.

**Process of Text Classification using Bi-LSTM**

The overall process starts with raw text data and ends with accurate predictions of text categories, typically involving the following steps:

1. **Data Preprocessing-**

The dataset is first cleaned (removing stop words, lowercasing, punctuation, etc.) and tokenized into sequences of integers. These sequences are padded to ensure consistent input length, enabling batch processing.

1. **Model Construction-**

A Sequential model is built starting with an Embedding layer. This is followed by a Bi-LSTM layer that processes each input sequence in both directions. A dropout layer is added for regularization, followed by a dense layer and a softmax/sigmoid output layer for classification.

1. **Training the Model-**

The model is compiled using optimizers like Adam and loss functions such as binary or categorical crossentropy, depending on the task. During training, the model adjusts its internal weights using backpropagation through time to minimize the loss.

1. **Evaluation and Prediction-**

The trained model is tested on a separate validation/test dataset. Predictions are generated and compared with actual labels to calculate performance metrics. The bidirectional architecture typically results in more accurate predictions, especially in context-heavy tasks.

**Model Evaluation and Visualization**

Performance is assessed using standard classification metrics to ensure that the model generalizes well to unseen data.

Performance Metrics:

* **Accuracy** – Measures the overall percentage of correct predictions.
* **Precision** – Indicates how many of the predicted positives are actually correct.
* **Recall** – Reflects how many actual positives the model correctly identified.
* **F1-Score** – Harmonic mean of precision and recall, useful in cases of class imbalance.

**Visualization of Results**

Accuracy and loss curves across epochs are plotted to assess learning behavior and convergence. Confusion matrices visualize true vs. predicted labels, making it easier to identify which classes are well-learned and which are misclassified.

**Conclusion**

The Bi-directional LSTM architecture effectively enhances model understanding of textual data by capturing both previous and future context, leading to better performance in classification tasks. Its dual-layer setup allows deeper semantic learning, resulting in improved accuracy and generalization. Evaluating through metrics and visual tools confirms the strength of Bi-LSTM over standard unidirectional models in natural language applications.

**Code and Outputs:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import classification\_report, confusion\_matrix

import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from tensorflow.keras.models import Sequential

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

df = pd.read\_csv('ecommerce\_datsaset.csv',usecols=['Text', 'label'])

df['Text'] = df['Text'].astype(str)

label\_counts = df['label'].value\_counts()

valid\_labels = label\_counts[label\_counts > 1].index

df = df[df['label'].isin(valid\_labels)]

label\_encoder = LabelEncoder()

df['label\_encoded'] = label\_encoder.fit\_transform(df['label'])

tokenizer = Tokenizer(oov\_token='<OOV>')

tokenizer.fit\_on\_texts(df['Text'])

sequences = tokenizer.texts\_to\_sequences(df['Text'])

word\_index = tokenizer.word\_index

max\_length = 100

X = pad\_sequences(sequences, maxlen=max\_length, padding='post')

y = df['label\_encoded'].values

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, stratify=y, random\_state=42)

# Model

vocab\_size = len(word\_index) + 1

embedding\_dim = 64

model = Sequential([

    Embedding(vocab\_size, embedding\_dim, input\_length=max\_length),

    Bidirectional(LSTM(64, return\_sequences=False)),

    Dropout(0.5),

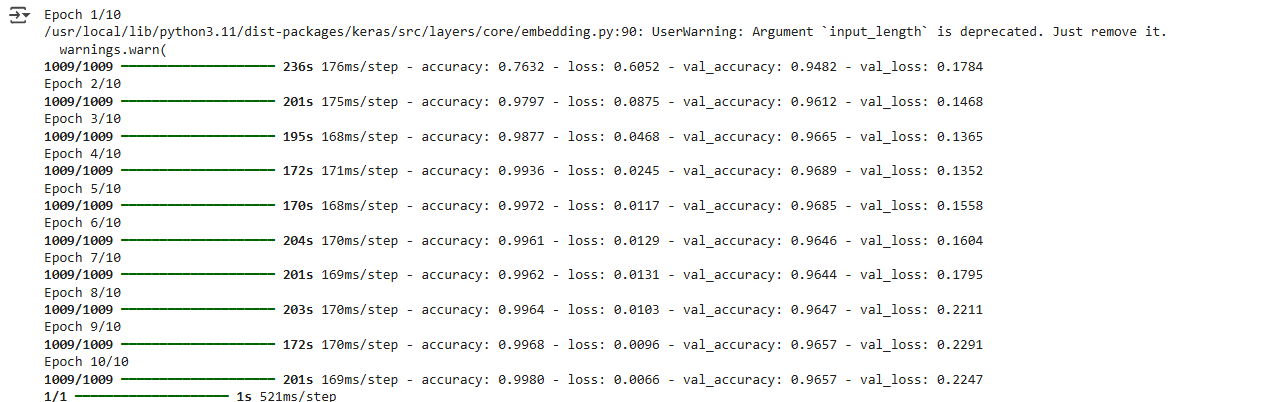
    Dense(32, activation='relu'),

    Dense(len(np.unique(y)), activation='softmax')

])

model.compile(loss='sparse\_categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)



# Sample predictions

sample\_texts = [

    "Bluetooth speaker with long battery life",

    "Casual summer dress for women",

    "Hardcover fiction novel by famous author",

    "Comfortable cotton bed sheets",

    "Smartwatch with fitness tracking",

    "Men's running shoes with arch support",

    "Designer handbag made from leather",

    "Children’s illustrated story book",

    "LED monitor for gaming setup",

    "Kitchen blender with multiple speed settings"

]

sample\_seq = tokenizer.texts\_to\_sequences(sample\_texts)

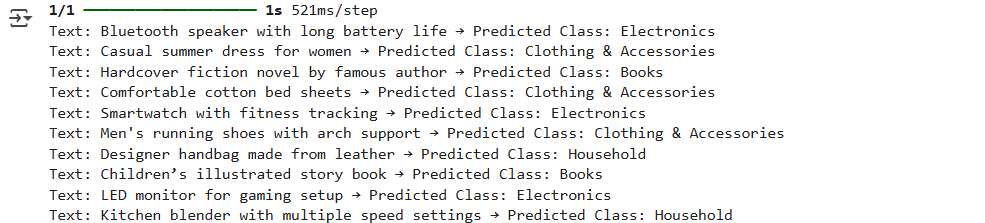
sample\_pad = pad\_sequences(sample\_seq, maxlen=max\_length, padding='post')

predictions = model.predict(sample\_pad)

for i, text in enumerate(sample\_texts):

    pred\_class = label\_encoder.inverse\_transform([np.argmax(predictions[i])])[0]

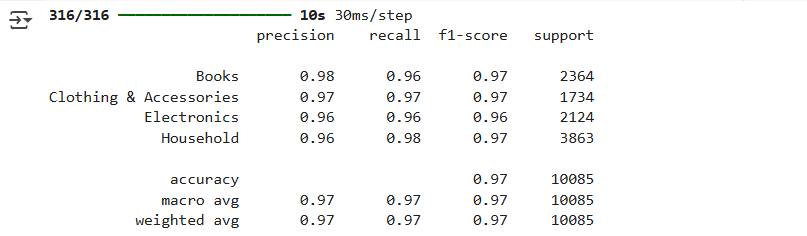
    print(f"Text: {text} → Predicted Class: {pred\_class}")



# Evaluation

y\_pred = np.argmax(model.predict(X\_test), axis=1)

print(classification\_report(y\_test, y\_pred, target\_names=label\_encoder.classes\_))



# Accuracy & loss plot

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

plt.subplot(1,2,1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Val Accuracy')

plt.legend()

plt.title('Accuracy over epochs')

plt.subplot(1,2,2)

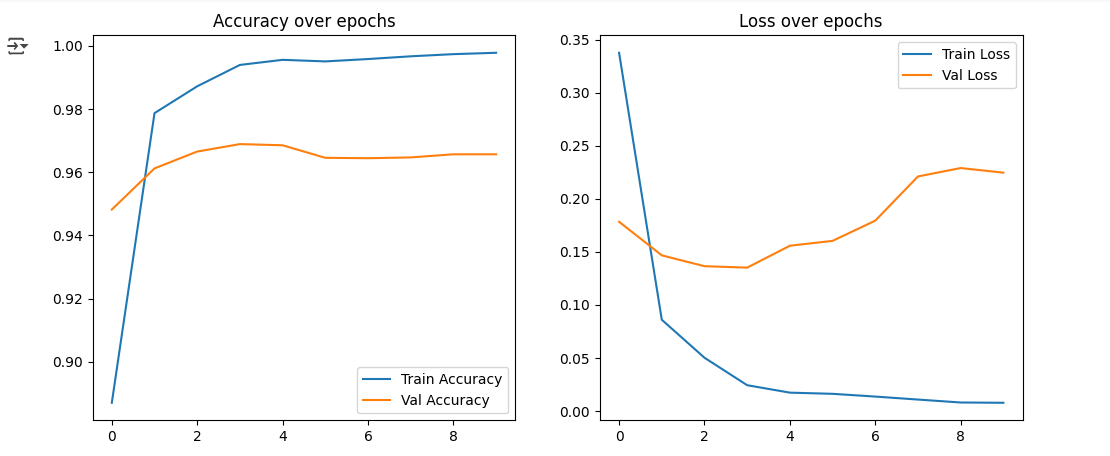
plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Val Loss')

plt.legend()

plt.title('Loss over epochs')

plt.show()

# Confusion matrix

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(8,6))

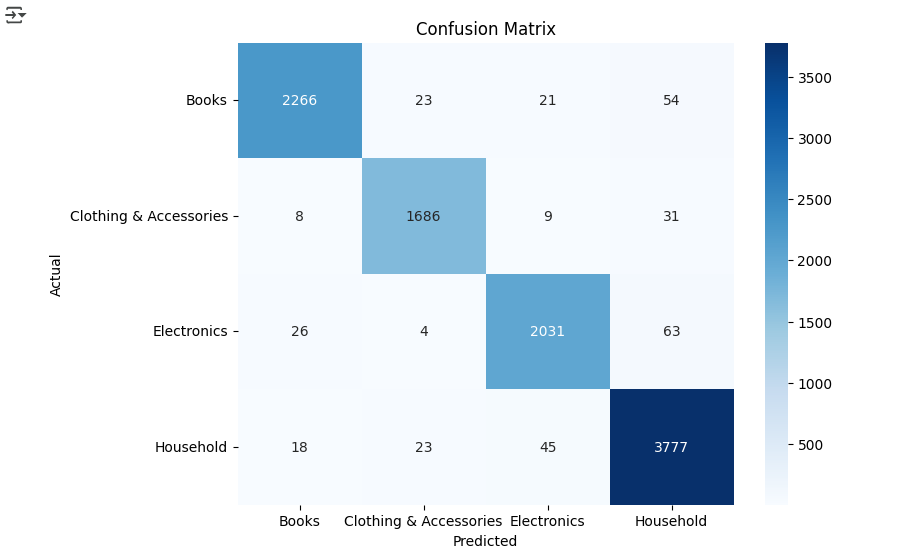
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label\_encoder.classes\_, yticklabels=label\_encoder.classes\_)

plt.xlabel('Predicted')

plt.ylabel('Actual')

plt.title('Confusion Matrix')

plt.show()



**Learning Outcomes:**

1. Understood the working of Bi-directional LSTM and its ability to capture context from both past and future sequences in text data.
2. Gained practical experience in building and training Bi-LSTM models for text classification using Python and deep learning frameworks.
3. Learned to evaluate model performance using metrics like accuracy, precision, recall, and F1-score for reliable classification results.