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

SCHOOL OF ENGINEERING & TECHNOLOGY

BTECH Programme: AI&DS

Course Title: Fundamentals of Deep Learning Lab

Course Code: AIDS304P

Submitted To: Dr. Dimple Tiwari
Submitted By: Vikram Ranjan
Enrolment no.: 09917711922

VISION OF INSTITUTE

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

MISSION OF INSTITUTE

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

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

INDEX

S.No	EXP.	Date	Marks			Remark	Updated Marks (If any)	Sign
			Laboratory Assessment (15 Marks)	Class Participation (5 Marks)	Viva (5 Marks)			
1								
2								
3								
4	.							
5								
6								

Experiment 1

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

Explaination:

Tensorflow: TensorFlow is an open-source machine learning framework developed by Google. It provides tools for building, training, and deploying deep learning models, with support for neural networks, natural language processing, and computer vision. TensorFlow supports both low-level operations for customization and high-level APIs like Keras for simplicity. It is widely used in research and production due to its scalability and compatibility across devices, from desktops to mobile and cloud environments.

Keras: Keras is a high-level API (application programming interface) built on Python that's integrated into the TensorFlow library. Keras is used to solve machine learning problems, particularly deep learning. It covers the entire machine learning workflow, from data processing to deployment. Keras is designed to be user-friendly and reduce cognitive load:

- Simple, consistent interfaces
- Clear error messages
- Modular and composable models
- Easy to extend

The 5-Step Model Life-Cycle

A model has a life-cycle, and this very simple knowledge provides the backbone for both modeling a dataset and understanding the tf.keras API.

The five steps in the life-cycle are as follows:

1. Define the model.
2. Compile the model.
3. Fit the model.
4. Evaluate the model.
5. Make predictions.

Sequential Model API (Simple)

The sequential model API is the simplest and is the API that I recommend, especially when getting started. It is referred to as "sequential" because it involves defining a Sequential class and adding layers to the model one by one in a linear manner, from input to output.

Functional Model API (Advanced)

The functional API is more complex but is also more flexible. It involves explicitly connecting the output of one layer to the input of another layer. Each connection is specified.

Code

MIP for binary classification

```
import numpy as np
from numpy import argmax
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, StandardScaler
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Load the dataset
path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/ionosphere.csv'
df = read_csv(path, header=None)
X, y = df.values[:, :-1], df.values[:, -1]
X = X.astype('float32')
y = LabelEncoder().fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
n_features = X_train.shape[1]
model = Sequential()
model.add(Dense(32, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
model.add(Dense(16, activation='relu', kernel_initializer='he_normal'))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=0)
```

```
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f'Test Accuracy: {acc:.3f}')
```

 # Make a prediction

 # Example row for prediction

```
row = np.array([[1,0 ,0.99539, -0.05889, 0.85243, 0.02306, 0.83398, -0.37708, 1, 0.03760,
    0.85243, -0.17755, 0.59755, -0.44945, 0.60536, -0.38223, 0.84356, -0.38542,
    0.58212, -0.32192, 0.56971, -0.29674, 0.36946, -0.47357, 0.56811, -0.51171,
    0.41078, -0.46168, 0.21266, -0.34090, 0.42267, -0.54487, 0.18641, -0.45300]])

row_scaled = scaler.transform(row)  # Scale the input row using the same scaler

yhat = model.predict(row_scaled)
print(f'Predicted: {yhat} (class={int(yhat[0] > 0.5)})')  # Binary class prediction (0 or 1)
```

Output

```
Test Accuracy: 0.879
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
Predicted: [[0.9882793]] (class=1)
```

MLP for multiclass classification

```
import numpy as np
from numpy import argmax
from pandas import read_csv
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense

# Load the dataset
path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/iris.csv'
df = read_csv(path, header=None)
X, y = df.values[:, :-1], df.values[:, -1]
X = X.astype('float32')
```

```
y = LabelEncoder().fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
n_features = X_train.shape[1]

# Define the model
model = Sequential()
model.add(Dense(10, activation='relu', kernel_initializer='he_normal', input_shape=(n_features,)))
model.add(Dense(8, activation='relu', kernel_initializer='he_normal'))
model.add(Dense(3, activation='softmax'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
model.fit(X_train, y_train, epochs=150, batch_size=32, verbose=0)

# Evaluate the model
loss, acc = model.evaluate(X_test, y_test, verbose=0)
print(f'Test Accuracy: {acc:.3f}')

# Make a prediction
row = np.array([[5.1, 3.5, 1.4, 0.2]])  # Convert to a 2D NumPy array
yhat = model.predict(row)
print(f'Predicted: {yhat} (class={argmax(yhat[0])})')
```

Output

```
(100, 4) (50, 4) (100,) (50,)
C:\Users\Dell\AppData\Roaming\Python\Python311\site-packages\
Do not pass an `input_shape`/`input_dim` argument to a layer.
`Input(shape)` object as the first layer in the model instead
  super().__init__(activity_regularizer=activity_regularizer,
Test Accuracy: 0.960
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 65ms/step
Predicted: [[0.94025403 0.05679062 0.00295545]] (class=0)
```

MLP for Regression

```
import tensorflow as tf
from numpy import sqrt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_regression
from pandas import read_csv

# Load the housing dataset
path = 'https://raw.githubusercontent.com/jbrownlee/Datasets/master/housing.csv'
df = read_csv(path, header=None)
X, y = df.values[:, :-1], df.values[:, -1]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Create the MLP model
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(32, activation='relu'),
    Dense(1)
])
# Compile the model
model.compile(optimizer='adam', loss='mse', metrics=['mae'])
model.fit(X_train, y_train, epochs=20, batch_size=32, validation_split=0.2)
loss, mae = model.evaluate(X_test, y_test)
print(f"Test Loss (MSE): {loss:.4f}")
print(f"Test MAE: {mae:.4f}")
rmse = sqrt(loss)
print(f'MSE: {loss:.3f}, RMSE: {rmse:.3f}')
# Make a prediction
row = [0.00632, 18.00, 2.310, 0, 0.5380, 6.5750, 65.20, 4.0900, 1, 296.0, 15.30, 396.90, 4.98]
```

```
row_scaled = scaler.transform([row])  # Ensure the input is scaled the same way as training data
yhat = model.predict(row_scaled)
print(f'Predicted: {yhat[0][0]:.3f}')
```

Output

```
Test Loss (MSE): 23.8467
Test MAE: 3.3176
MSE: 23.847, RMSE: 4.883
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 55ms/step
Predicted: 31.206
```

Learning Outcome

Understand and utilize the basic features of TensorFlow and Keras packages in Python for building and training machine learning models. Gain hands-on experience in implementing neural networks using these libraries.

Experiment 2

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

Explaination:

ANN Regression: Artificial Neural Networks (ANNs) can be used for regression tasks by mapping input features to continuous output values. In Python, MLPRegressor from sklearn.neural_network is commonly used. The model consists of an input layer, hidden layers with activation functions (like ReLU), and an output layer with a linear activation function. The model is trained using backpropagation and gradient descent to minimize the Mean Squared Error (MSE) loss.

ANN for Classification: For classification problems, ANNs use MLPClassifier from sklearn.neural_network. The model maps input features to discrete class labels using softmax (multi-class) or sigmoid (binary) activation in the output layer. The loss function used is Cross-Entropy, and training is performed using stochastic gradient descent or Adam optimizer.

Code

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neural_network import MLPClassifier, MLPRegressor
from sklearn.metrics import accuracy_score, mean_squared_error, confusion_matrix, roc_curve, roc_auc_score
import matplotlib.pyplot as plt
import seaborn as sns
df_regression = pd.read_csv(r"C:\Users\Dell\Downloads\Real_Combine.csv")
df_regression.isnull().sum()
```

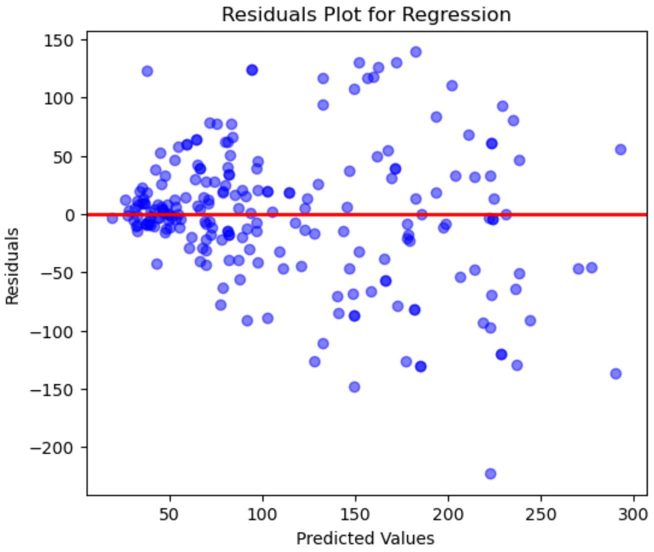
```
[3]: T         0
     TM        0
     Tm        0
     SLP       0
     H         0
     VV        0
     V         0
     VM        0
     PM 2.5    1
     dtype: int64
```

```
df_regression=df_regression.dropna()
# Regression task
X_regression=df_regression.iloc[:,:-1] ## independent features
y_regression=df_regression.iloc[:,-1] ## dependent features

# Split data into training and testing sets
```

```
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_regression, y_regression, test_size=0.2, random_state=42)

# Scaling the features for ANN (Standardize them)
scaler_reg = StandardScaler()
X_train_reg = scaler_reg.fit_transform(X_train_reg)
X_test_reg = scaler_reg.transform(X_test_reg)
model_reg = MLPRegressor(hidden_layer_sizes=(10,), max_iter=1000, random_state=42)
model_reg.fit(X_train_reg, y_train_reg)
y_pred_reg = model_reg.predict(X_test_reg)
mse = mean_squared_error(y_test_reg, y_pred_reg)
print(f'Regression Model MSE: {mse}')
```

```
Regression Model MSE: 3245.1890890604805
```

```
plt.figure(figsize=(6, 5))
plt.scatter(y_test_reg, y_pred_reg, color='blue', alpha=0.5)
plt.plot([min(y_test_reg), max(y_test_reg)], [min(y_test_reg), max(y_test_reg)], color='red', lw=2)  # Ideal line
plt.title('Predicted vs Actual for Regression')
plt.xlabel('Actual Values')
plt.ylabel('Predicted Values')
plt.show()
```

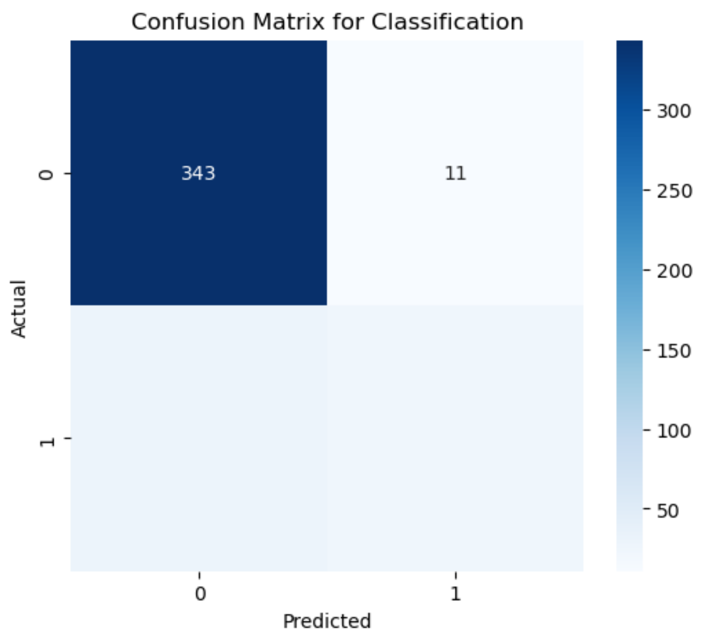



```
# Classification task
df_classification = pd.read_csv(r"C:\Users\Dell\Desktop\data.csv")
```

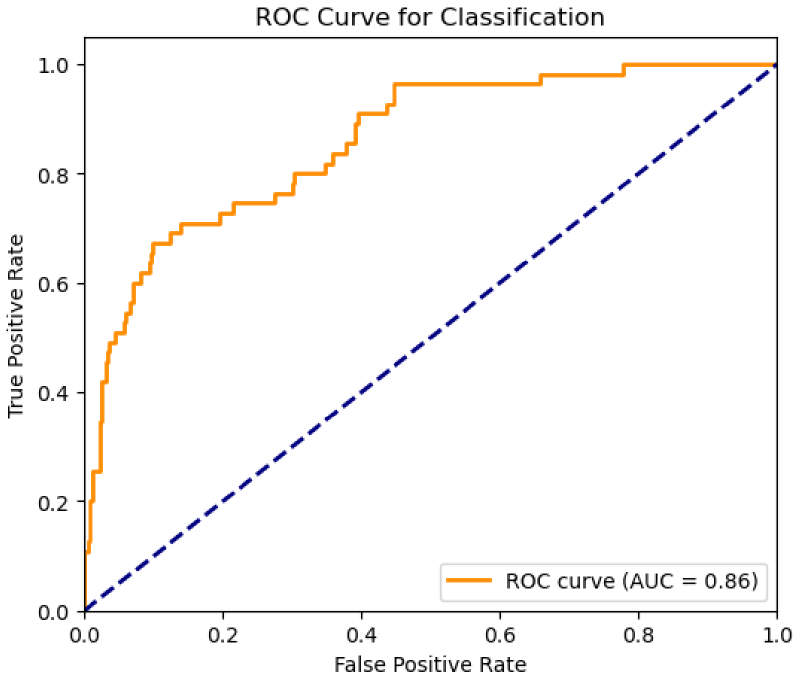
```
X_classification = df_classification.iloc[:,:-2] ## independent features
y_classification = df_classification.iloc[:,-2] ## dependent features

# Split data into training and testing sets
X_train_class, X_test_class, y_train_class, y_test_class = train_test_split(X_classification, y_classification, test_size=0.2, random_state=42)
# Scaling the features for ANN (Standardize them)
scaler_class = StandardScaler()
X_train_class = scaler_class.fit_transform(X_train_class)
X_test_class = scaler_class.transform(X_test_class)

# Create and train the ANN classifier
model_class = MLPClassifier(hidden_layer_sizes= (10,), max_iter=1000, random_state=42)
model_class.fit(X_train_class, y_train_class)
y_pred_class = model_class.predict(X_test_class)
accuracy = accuracy_score(y_test_class, y_pred_class)
print(f'Classification Model Accuracy: {accuracy}')
```

```
Classification Model Accuracy: 0.8997555012224939
```

```
cm = confusion_matrix(y_test_class, y_pred_class)
plt.figure(figsize=(6, 5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=[0, 1], yticklabels=[0, 1])
plt.title('Confusion Matrix for Classification')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()

fpr, tpr, thresholds = roc_curve(y_test_class, model_class.predict_proba(X_test_class)[:, 1])
roc_auc = roc_auc_score(y_test_class, model_class.predict_proba(X_test_class)[:, 1])
```



```
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Classification')
plt.legend(loc='lower right')
plt.show()
```



Learning Outcome:

Understand how to implement ANN models for both regression and classification using Python. Gain insights into choosing appropriate activation functions and loss functions based on the problem type.

Experiment 3

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

Explaination:

CNN A Convolutional Neural Network is a class of artificial neural network that uses convolutional layers to filter inputs for useful information. The convolution operation involves combining input data (feature map) with a convolution kernel (filter) to form a transformed feature map. The filters in the convolutional layers (conv layers) are modified based on learned parameters to extract the most useful information for a specific task. Convolutional networks adjust automatically to find the best feature based on the task. The CNN would filter information about the shape of an object when confronted with a general object recognition task but would extract the color of the bird when faced with a bird recognition task. This is based on the CNN's understanding that different classes of objects have different shapes but that different types of birds are more likely to differ in color than in shape.

Code

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from sklearn.model_selection import train_test_split
from tensorflow.keras.utils import to_categorical

import tensorflow as tf
import numpy as np
import os
from tensorflow.keras.preprocessing.image import load_img, img_to_array

# Define dataset path
data_dir = "D:/FDL file/Dataset/Brain Tumor MRI images"
categories = ["Tumor", "No_Tumor"]
IMG_SIZE = 128  # Resize to 128x128
```

```
data = []
labels = []

for category in categories:
    path = os.path.join(data_dir, category)
    label = categories.index(category)  # 0 for No_Tumor, 1 for Tumor

    for img_name in os.listdir(path):
        try:
            img_path = os.path.join(path, img_name)
            img = load_img(img_path, color_mode="grayscale", target_size=(IMG_SIZE, IMG_SIZE))
            img_array = img_to_array(img) / 255.0  # Normalize
            data.append(img_array)
            labels.append(label)
        except Exception as e:
            print(f"Error loading {img_name}: {e}")

# Convert to NumPy arrays
data = np.array(data).reshape(-1, IMG_SIZE, IMG_SIZE, 1)
labels = np.array(labels)

# Split dataset into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2, random_state=42)

# Convert labels to categorical (one-hot encoding)
y_train = to_categorical(y_train, num_classes=2)
y_test = to_categorical(y_test, num_classes=2)

# Define CNN model
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 1)),
    MaxPooling2D(pool_size=(2, 2)),
```

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

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

    Flatten(),
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(2, activation='softmax')  # Output layer (2 classes: Tumor, No Tumor)
])

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

# Print model summary
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 126, 126, 32)	320
max_pooling2d (MaxPooling2D)	(None, 63, 63, 32)	0
conv2d_1 (Conv2D)	(None, 61, 61, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 30, 30, 64)	0
conv2d_2 (Conv2D)	(None, 28, 28, 128)	73,856
max_pooling2d_2 (MaxPooling2D)	(None, 14, 14, 128)	0
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 128)	3,211,392
dropout (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 2)	258

Total params: 3,304,322 (12.60 MB)

Trainable params: 3,304,322 (12.60 MB)

Non-trainable params: 0 (0.00 B)

```
# Train the model
```

history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_test, y_test))

```
Executing op __inference_multi_step_on_iterator_4338 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op __inference_multi_step_on_iterator_4338 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op __inference_multi_step_on_iterator_4338 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op AnonymousIteratorV3 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op MakeIterator in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op DivNoNan in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op DivNoNan in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
125/125 ━━━━━━━━━━━━━━━━━━━━ 10s 81ms/step - accuracy: 0.9989 - loss: 0.0053 - val_accuracy: 0.9750 - val_loss: 0.1162
```

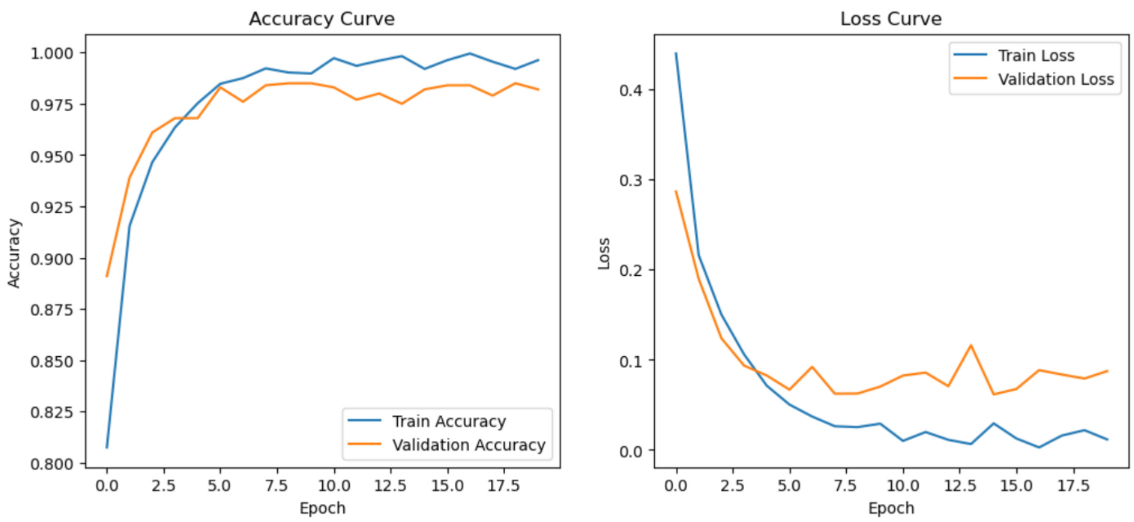
Evaluate on test set
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_acc:.4f}")

```
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
30/32 ━━━━━━━━━━━━━━━━━━━━ 0s 21ms/step - accuracy: 0.9822 - loss: 0.0810Executing op __inference_multi_step_on_iterator_4338 in device /job:localhost/
replica:0/task:0/device:CPU:0
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op __inference_multi_step_on_iterator_4338 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalHasValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op OptionalGetValue in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op AnonymousIteratorV3 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op MakeIterator in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op DivNoNan in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op ReadVariableOp in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op DivNoNan in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op _EagerConst in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Range in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op Mean in device /job:localhost/replica:0/task:0/device:CPU:0
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 21ms/step - accuracy: 0.9822 - loss: 0.0817
Test Accuracy: 0.9820
```

# Plot training accuracy and loss
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='Validation
Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Curve')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')

plt.show()
```

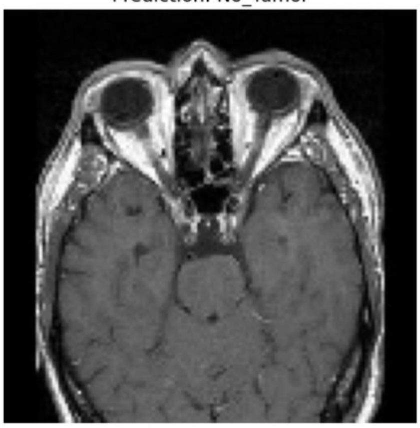



```
from tensorflow.keras.preprocessing.image import load_img,
img_to_array
import numpy as np
import matplotlib.pyplot as plt

# Function to predict a new MRI image without OpenCV
def predict_image(image_path):
    img = load_img(image_path, color_mode="grayscale",
target_size=(IMG_SIZE, IMG_SIZE))  # Load and resize
    img_array = img_to_array(img) / 255.0  # Normalize
    img_reshaped = img_array.reshape(1, IMG_SIZE,
IMG_SIZE, 1)  # Reshape for model input
```

```
    prediction = model.predict(img_reshaped)
    predicted_label = categories[np.argmax(prediction)]  # Get
class label

    plt.imshow(img, cmap='gray')
    plt.title(f"Prediction: {predicted_label}")
    plt.axis("off")  # Hide axes for better visualization
    plt.show()


# Example usage
predict_image("D:/FDL file/Dataset/Brain Tumor MRI
images/No_Tumor/mri_healthy (1).jpg")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 105ms/stepExecuting op AnonymousIteratorV3 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op MakeIterator in device /job:localhost/replica:0/task:0/device:CPU:0
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 126ms/step
```

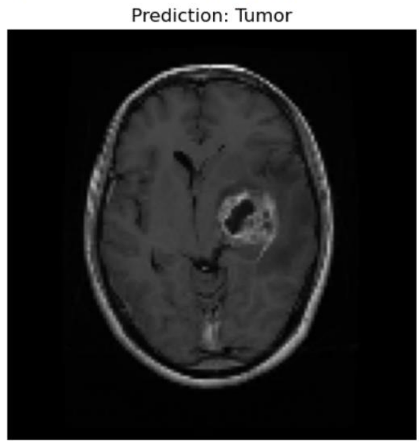
Prediction: No_Tumor



```
from tensorflow.keras.preprocessing.image import load_img,
img_to_array
import numpy as np
import matplotlib.pyplot as plt

# Function to predict a new MRI image without OpenCV
def predict_image(image_path):
    img = load_img(image_path, color_mode="grayscale",
target_size=(IMG_SIZE, IMG_SIZE))  # Load and resize
    img_array = img_to_array(img) / 255.0  # Normalize
    img_reshaped = img_array.reshape(1, IMG_SIZE,
IMG_SIZE, 1)  # Reshape for model input

    prediction = model.predict(img_reshaped)
```

```
    predicted_label = categories[np.argmax(prediction)]  # Get
class label

    plt.imshow(img, cmap='gray')
    plt.title(f"Prediction: {predicted_label}")
    plt.axis("off")  # Hide axes for better visualization
    plt.show()

# Example usage
predict_image("D:/FDL file/Dataset/Brain Tumor MRI
images/Tumor/glioma (18).jpg")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 32ms/stepExecuting op AnonymousIteratorV3 in device /job:localhost/replica:0/task:0/device:CPU:0
Executing op MakeIterator in device /job:localhost/replica:0/task:0/device:CPU:0
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step
```



Learning Outcome:

- Learn how CNNs extract features from images using convolutions, pooling, and fully connected layers.
- Develop a custom CNN architecture for classification.
- Visualizing results using Matplotlib.

Experiment 4

Objective: Implementation of Autoencoders for dimensionality reduction in Python.

Explanation:

Autoencoders are unsupervised neural networks used for dimensionality reduction by learning efficient data representations. They consist of an encoder that compresses input data into a latent space and a decoder that reconstructs the original input. This helps retain essential features while removing noise and redundancy. Implemented in Python using TensorFlow/Keras, autoencoders train by minimizing reconstruction loss, making them useful for feature extraction, anomaly detection, and noise reduction. Unlike traditional techniques like PCA, autoencoders can capture nonlinear relationships. By reducing dimensions, they enhance computational efficiency, aiding machine learning tasks while preserving crucial information in high-dimensional datasets.

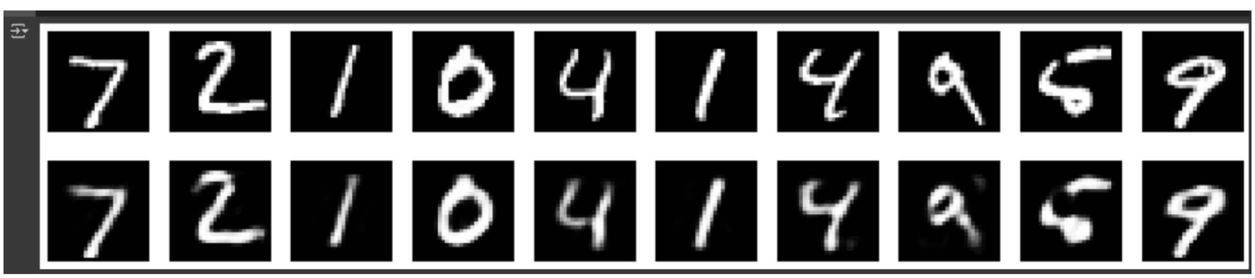
Code & Output:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Dense
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt
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
x_train = x_train.reshape((len(x_train), -1))  # Flatten images
x_test = x_test.reshape((len(x_test), -1))

# Define encoding dimension
```

```
encoding_dim = 32  # Reduced dimension


# Encoder
input_img = Input(shape=(784,))
encoded = Dense(encoding_dim, activation='relu')(input_img)


# Decoder
decoded = Dense(784, activation='sigmoid')(encoded)
# Autoencoder model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
# Train the model
autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True, validation_data=(x_test, x_test))


# Extract encoder model
encoder = Model(input_img, encoded)
```

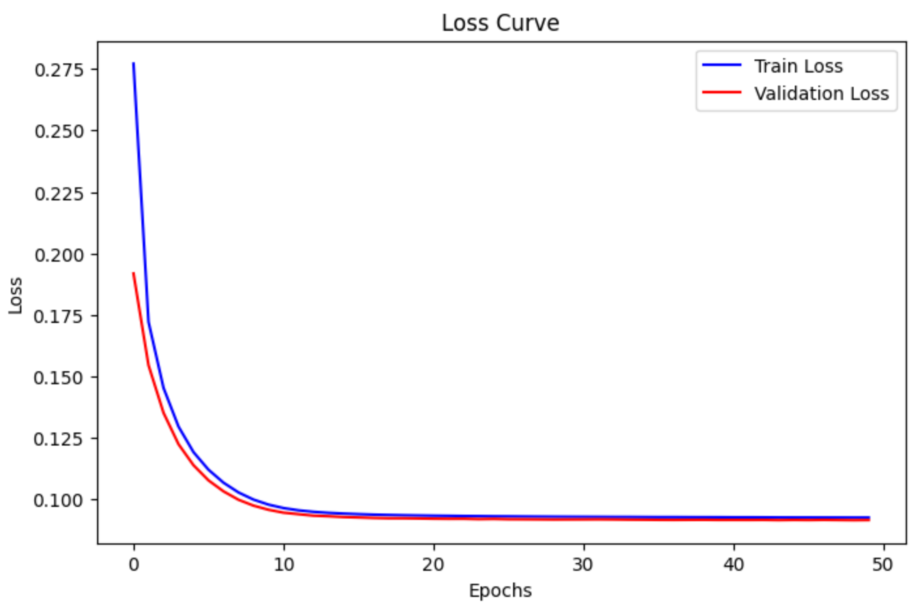
```
Epoch 1/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - loss: 0.3817 - val_loss: 0.1918
Epoch 2/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.1816 - val_loss: 0.1546
Epoch 3/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1500 - val_loss: 0.1352
Epoch 4/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1327 - val_loss: 0.1224
Epoch 5/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.1214 - val_loss: 0.1139
Epoch 6/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.1135 - val_loss: 0.1077
Epoch 7/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.1080 - val_loss: 0.1033
Epoch 8/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.1037 - val_loss: 0.0999
Epoch 9/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 8ms/step - loss: 0.1005 - val_loss: 0.0975
Epoch 10/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - loss: 0.0982 - val_loss: 0.0958
Epoch 11/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - loss: 0.0968 - val_loss: 0.0946
Epoch 12/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 3s 11ms/step - loss: 0.0958 - val_loss: 0.0940
```

```
# Encode test images
encoded_imgs = encoder.predict(x_test)

# Display original and reconstructed images
decoded_imgs = autoencoder.predict(x_test)

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

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



```
# Plot training and validation loss in one curve
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label='Train Loss', color='blue')
```

```
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')
plt.show()
```



Learning Outcome(s):

- Successfully implemented autoencoders to effectively reduce dimensionality while preserving essential features, enabling efficient data compression and reconstruction.
- Visualizing results using Matplotlib.

Experiment 5

Objective: Application of Autoencoders on Image Dataset.

Explanation:

When applied to image datasets, autoencoders learn to compress images into a lower-dimensional latent space and reconstruct them with minimal loss. This technique is useful in applications like image denoising, dimensionality reduction, and generative modeling. In Python, libraries like TensorFlow and Keras facilitate autoencoder implementation, allowing efficient training on datasets like MNIST. By minimizing reconstruction loss, autoencoders help in capturing essential features, making them valuable for various deep learning applications in image processing and computer vision.

Code & Output:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Dense, Flatten, Reshape
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10

# Load CIFAR-10 dataset
(x_train, _), (x_test, _) = cifar10.load_data()
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0
# Flatten images
x_train = x_train.reshape((len(x_train), -1))
x_test = x_test.reshape((len(x_test), -1))
```

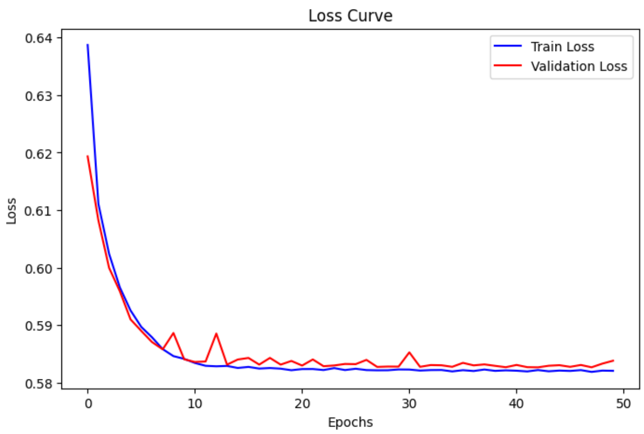
```
# Define encoding dimension
encoding_dim = 128  # Increased dimension for CIFAR-10

# Encoder
input_img = Input(shape=(3072,))  # CIFAR-10 images are 32x32x3
encoded = Dense(encoding_dim, activation='relu')(input_img)

# Decoder
decoded = Dense(3072, activation='sigmoid')(encoded)

# Autoencoder model
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
history = autoencoder.fit(x_train, x_train, epochs=50, batch_size=256, shuffle=True, validation_data=(x_test, x_test))

# Extract encoder model
encoder = Model(input_img, encoded)

# Encode test images
encoded_imgs = encoder.predict(x_test)
```

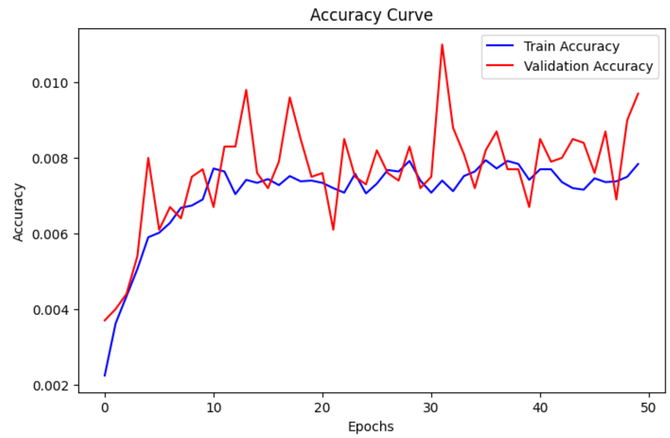
```
Epoch 1/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 13s 60ms/step - accuracy: 0.0017 - loss: 0.6593 - val_accuracy: 0.0037 - val_loss: 0.6193
Epoch 2/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 11s 55ms/step - accuracy: 0.0035 - loss: 0.6144 - val_accuracy: 0.0040 - val_loss: 0.6082
Epoch 3/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 19s 49ms/step - accuracy: 0.0045 - loss: 0.6039 - val_accuracy: 0.0044 - val_loss: 0.6000
Epoch 4/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 12s 58ms/step - accuracy: 0.0047 - loss: 0.5977 - val_accuracy: 0.0054 - val_loss: 0.5959
Epoch 5/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 11s 55ms/step - accuracy: 0.0061 - loss: 0.5934 - val_accuracy: 0.0080 - val_loss: 0.5910
Epoch 6/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 21s 57ms/step - accuracy: 0.0063 - loss: 0.5906 - val_accuracy: 0.0061 - val_loss: 0.5890
Epoch 7/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 19s 52ms/step - accuracy: 0.0062 - loss: 0.5883 - val_accuracy: 0.0067 - val_loss: 0.5871
Epoch 8/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 11s 55ms/step - accuracy: 0.0067 - loss: 0.5861 - val_accuracy: 0.0064 - val_loss: 0.5858
Epoch 9/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 11s 57ms/step - accuracy: 0.0065 - loss: 0.5853 - val_accuracy: 0.0075 - val_loss: 0.5886
```

```
# Display original and reconstructed images
decoded_imgs = autoencoder.predict(x_test)

n = 10  # Number of images to display
plt.figure(figsize=(20, 4))
for i in range(n):
    # Original images
    ax = plt.subplot(2, n, i + 1)
    plt.imshow(x_test[i].reshape(32, 32, 3))
    plt.axis('off')

    # Reconstructed images
    ax = plt.subplot(2, n, i + 1 + n)
    plt.imshow(decoded_imgs[i].reshape(32, 32, 3))
    plt.axis('off')
plt.show()
```

```
# Plot training and validation loss
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label='Train Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')
plt.show()

# Handle possible metric naming issues
train_acc_key = 'accuracy' if 'accuracy' in history.history else 'binary_accuracy'
val_acc_key = 'val_accuracy' if 'val_accuracy' in history.history else 'val_binary_accuracy'

# Plot training and validation accuracy
plt.figure(figsize=(8, 5))
plt.plot(history.history[train_acc_key], label='Train Accuracy', color='blue')
plt.plot(history.history[val_acc_key], label='Validation Accuracy', color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
plt.title('Accuracy Curve')
plt.show()
```





Learning Outcome(s):

- Successfully implemented autoencoders on image data in Python.
- Visualizing results using Matplotlib.

Experiment 6

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

Explanation:

Convolutional autoencoders (CAEs) improve standard autoencoders by utilizing convolutional layers, making them more efficient in capturing spatial hierarchies in image data. Unlike fully connected autoencoders, CAEs preserve local spatial structures, reducing redundancy while enhancing feature extraction. When applied to the MNIST dataset, convolutional layers help the network learn key image patterns, leading to better reconstruction with fewer parameters. Implementing CAEs in Python using TensorFlow and Keras involves replacing dense layers with convolutional and pooling layers, resulting in improved performance, reduced loss, and sharper reconstructed images. This approach is widely used in noise removal, anomaly detection, and feature learning tasks.

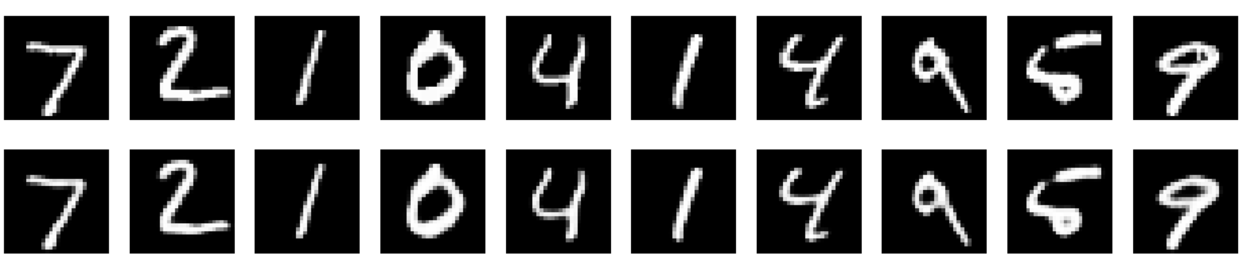
Code & Output:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Input, Conv2D, MaxPooling2D, UpSampling2D
from tensorflow.keras.models import Model
import numpy as np
import matplotlib.pyplot as plt
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
x_train = np.expand_dims(x_train, axis=-1)  # Add channel dimension
x_test = np.expand_dims(x_test, axis=-1)
```

```
# Define Convolutional Autoencoder
input_img = Input(shape=(28, 28, 1))

# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)

autoencoder = Model(input_img, x)
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')

# Train the model
history = autoencoder.fit(x_train, x_train, epochs=20, batch_size=256, shuffle=True,
validation_data=(x_test, x_test))

# Encode and decode images
decoded_imgs = autoencoder.predict(x_test)
```

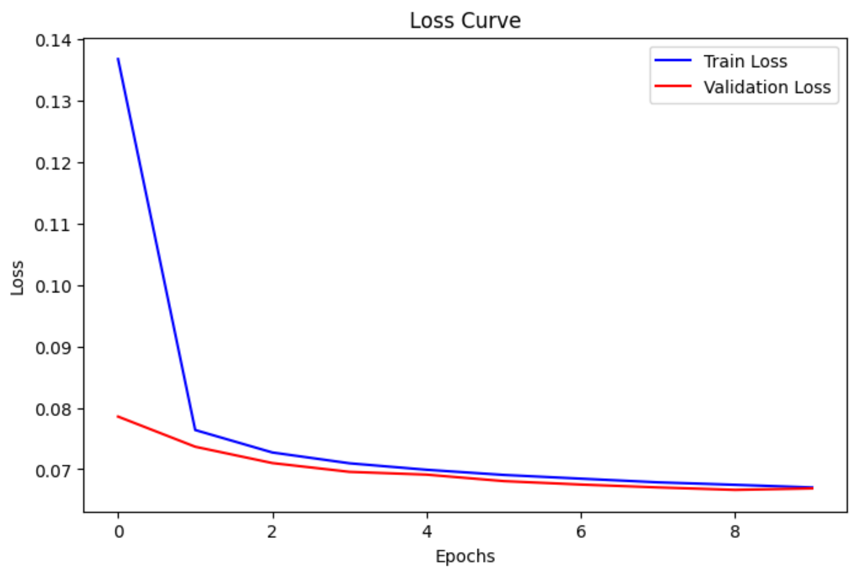
```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 169s 708ms/step - loss: 0.2436 - val_loss: 0.0786
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 202s 708ms/step - loss: 0.0779 - val_loss: 0.0737
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 199s 694ms/step - loss: 0.0734 - val_loss: 0.0710
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 205s 708ms/step - loss: 0.0713 - val_loss: 0.0696
Epoch 5/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 171s 728ms/step - loss: 0.0702 - val_loss: 0.0691
Epoch 6/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 198s 709ms/step - loss: 0.0694 - val_loss: 0.0681
Epoch 7/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 169s 719ms/step - loss: 0.0685 - val_loss: 0.0675
Epoch 8/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 195s 692ms/step - loss: 0.0680 - val_loss: 0.0670
Epoch 9/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 168s 716ms/step - loss: 0.0676 - val_loss: 0.0666
Epoch 10/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 202s 715ms/step - loss: 0.0671 - val_loss: 0.0669
```

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



```
# Plot training and validation loss
plt.figure(figsize=(8, 5))
```

```
plt.plot(history.history['loss'], label='Train Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')
plt.show()
```



Learning Outcome(s):

- Successfully improved Autocoder's Performance using convolution layers in Python
- Visualizing results using Matplotlib.

Experiment 7

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

Explanation:

Recurrent Neural Networks (RNNs) are widely used for time-series forecasting, including stock price prediction. RNNs process sequential data by maintaining a memory of past inputs, making them ideal for capturing trends and patterns in stock market data. In Python, TensorFlow and Keras provide tools to build RNN models using layers like SimpleRNN, LSTM, or GRU. The model is trained on historical stock prices, learning dependencies over time. By predicting future prices based on past trends, RNNs help in financial analysis and decision-making. Proper tuning of hyperparameters, such as the number of layers and epochs, enhances prediction accuracy and model efficiency.

Code & Output:

```
import numpy as np

import pandas as pd

import yfinance as yf

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.metrics import MeanAbsolutePercentageError


# Load stock price data (Apple - AAPL)

stock_symbol = 'AAPL'

data = yf.download(stock_symbol, start='2020-01-01', end='2024-01-01')


# Use 'Close' price for prediction
```

```
prices = data['Close'].values.reshape(-1, 1)

# Normalize data
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_prices = scaler.fit_transform(prices)

# Create dataset sequences
def create_dataset(data, time_steps=10):
    X, y = [], []
    for i in range(time_steps, len(data)):
        X.append(data[i - time_steps:i, 0])
        y.append(data[i, 0])
    return np.array(X), np.array(y)

time_steps = 10
X, y = create_dataset(scaled_prices, time_steps)

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

# Train-test split
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Build the RNN model
```

model = Sequential([

SimpleRNN(50, activation='relu', return_sequences=True, input_shape=(X_train.shape[1], 1)),

Dropout(0.2),

SimpleRNN(50, activation='relu'),

Dropout(0.2),

Dense(1)

])

Compile model

model.compile(optimizer=Adam(learning_rate=0.001), loss='mean_squared_error', metrics=[MeanAbsolutePercentageError()])

Train the model

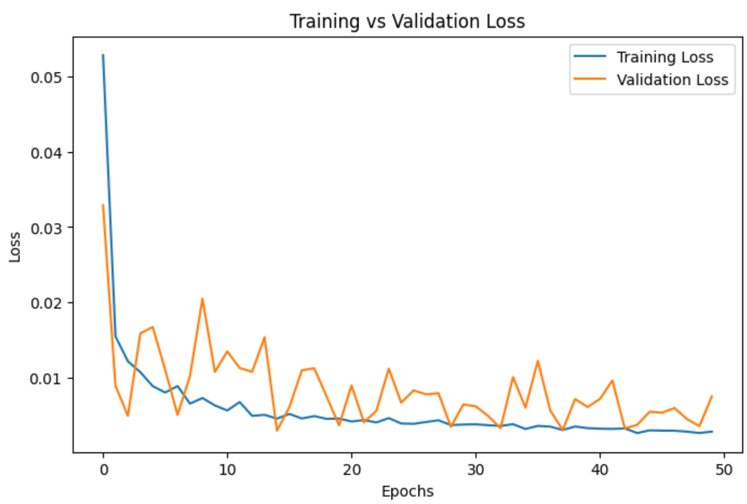
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test))

```
Epoch 1/50
/usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input_shape`/`inpu
  super().__init__(**kwargs)
25/25 ━━━━━━━━━━━━━━━━━━━━ 5s 33ms/step - loss: 0.1009 - val_loss: 0.0329
Epoch 2/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0187 - val_loss: 0.0089
Epoch 3/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - loss: 0.0134 - val_loss: 0.0049
Epoch 4/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - loss: 0.0121 - val_loss: 0.0159
Epoch 5/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 1s 10ms/step - loss: 0.0082 - val_loss: 0.0167
Epoch 6/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0077 - val_loss: 0.0110
Epoch 7/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0087 - val_loss: 0.0050
Epoch 8/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0066 - val_loss: 0.0102
Epoch 9/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0073 - val_loss: 0.0205
Epoch 10/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - loss: 0.0067 - val_loss: 0.0108
Epoch 11/50
25/25 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - loss: 0.0055 - val_loss: 0.0135
```

Plot Training & Validation Loss

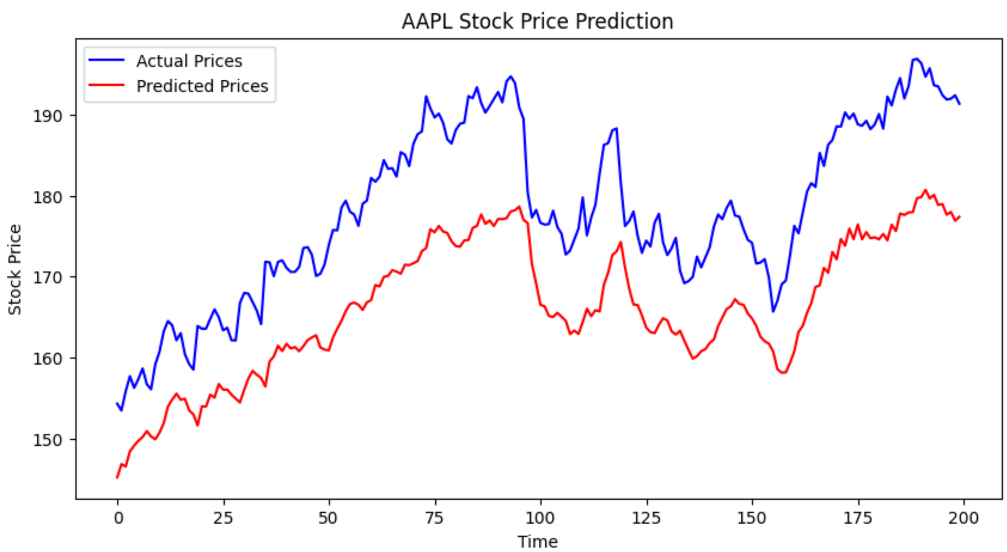
plt.figure(figsize=(8,5))

```
plt.plot(history.history['loss'], label='Training Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training vs Validation Loss')
plt.legend()
plt.show()
```




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

# Inverse transform predictions
y_pred = scaler.inverse_transform(y_pred.reshape(-1, 1))
y_test_actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plot Actual vs Predicted Prices
plt.figure(figsize=(10,5))
plt.plot(y_test_actual, label='Actual Prices', color='blue')
```

```
plt.plot(y_pred, label='Predicted Prices', color='red')
plt.xlabel('Time')
plt.ylabel('Stock Price')
plt.title(f'{stock_symbol} Stock Price Prediction')
plt.legend()
plt.show()
```



Learning Outcome(s):

- Successfully implemented RNN model for Stock Price Prediction in Python.
- Visualizing results using Matplotlib.

Experiment 8

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

Explanation:

Long Short-Term Memory (LSTM) networks are a specialized form of Recurrent Neural Networks (RNNs) that excel in time-series forecasting, making them ideal for predicting future weather patterns in cities. LSTMs retain long-term dependencies through memory cells, allowing them to learn seasonal trends and variations in weather data. In Python, TensorFlow and Keras enable building an LSTM model trained on historical weather data, including temperature, humidity, and wind speed. The model learns temporal relationships and forecasts future conditions based on past trends. Proper data preprocessing, hyperparameter tuning, and feature selection significantly impact prediction accuracy and reliability in real-world applications.

Code & Output:

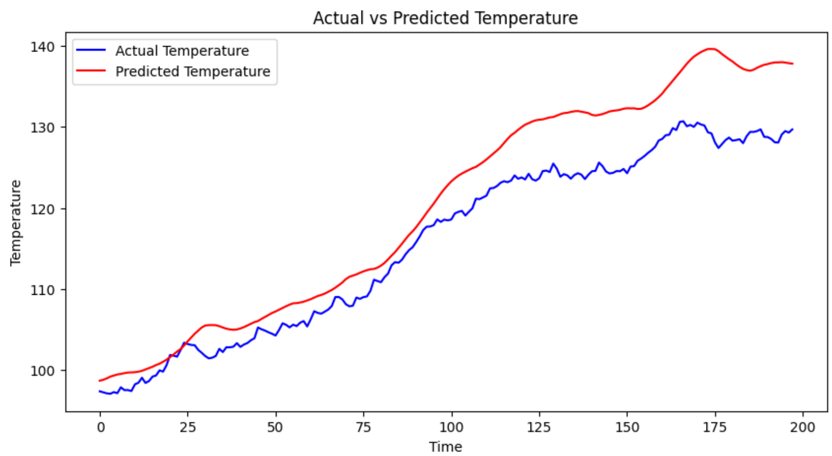
```
import numpy as np

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

from sklearn.preprocessing import MinMaxScaler


# Generate synthetic weather data (temperature) for demonstration

np.random.seed(42)

temp_data = np.cumsum(np.random.randn(1000) * 0.5 + 0.1) + 20  # Simulated
temperature data


# Prepare dataset

def create_dataset(data, time_steps=10):
```

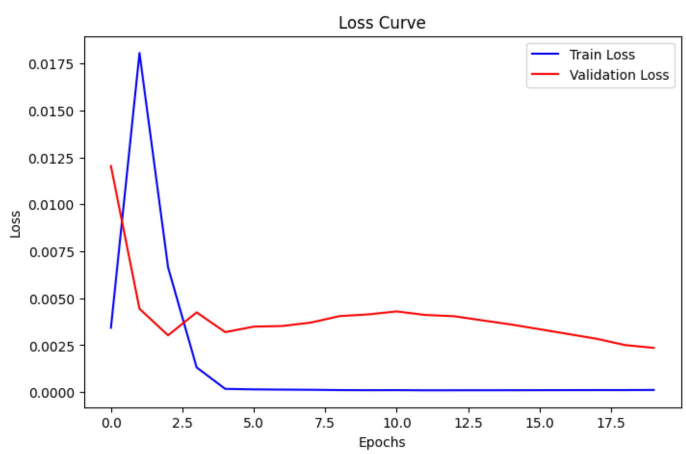
```python
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps])
        y.append(data[i+time_steps])
    return np.array(X), np.array(y)
scaler = MinMaxScaler()
temp_data_scaled = scaler.fit_transform(temp_data.reshape(-1, 1))
time_steps = 10
X, y = create_dataset(temp_data_scaled, time_steps)
X = np.reshape(X, (X.shape[0], X.shape[1], 1))  # Reshaping for LSTM
# Split into training and testing sets
split = int(0.8 * len(X))
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

# Define LSTM model
model = Sequential([
    LSTM(50, activation='relu', return_sequences=True, input_shape=(time_steps, 1)),
    LSTM(50, activation='relu'),
    Dense(1)
])
model.compile(optimizer='adam', loss='mse')

# Train the model
history = model.fit(X_train, y_train, epochs=20, batch_size=16, validation_data=(X_test,
y_test), shuffle=False)
```

```
Epoch 1/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 9s 25ms/step - loss: 0.0034 - val_loss: 0.0120
Epoch 2/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 0.0321 - val_loss: 0.0044
Epoch 3/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0155 - val_loss: 0.0030
Epoch 4/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 0.0033 - val_loss: 0.0042
Epoch 5/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 15ms/step - loss: 2.3305e-04 - val_loss: 0.0032
Epoch 6/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - loss: 2.2920e-04 - val_loss: 0.0035
Epoch 7/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 16ms/step - loss: 1.5116e-04 - val_loss: 0.0035
Epoch 8/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 1.7497e-04 - val_loss: 0.0037
Epoch 9/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 2s 19ms/step - loss: 1.2475e-04 - val_loss: 0.0040
Epoch 10/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 22ms/step - loss: 1.0340e-04 - val_loss: 0.0041
Epoch 11/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - loss: 1.1726e-04 - val_loss: 0.0043
Epoch 12/20
50/50 ━━━━━━━━━━━━━━━━━━━━ 1s 14ms/step - loss: 8.6570e-05 - val_loss: 0.0041
```

```
# Predict and inverse transform
predicted_temp = model.predict(X_test)
predicted_temp = scaler.inverse_transform(predicted_temp)
y_test_actual = scaler.inverse_transform(y_test.reshape(-1, 1))

# Plot actual vs predicted temperatures
plt.figure(figsize=(10, 5))
plt.plot(y_test_actual, label='Actual Temperature', color='blue')
plt.plot(predicted_temp, label='Predicted Temperature', color='red')
plt.xlabel('Time')
plt.ylabel('Temperature')
plt.legend()
plt.title('Actual vs Predicted Temperature')
plt.show()

# Plot training and validation loss
plt.figure(figsize=(8, 5))
```

```
plt.plot(history.history['loss'], label='Train Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')
plt.show()
```





Learning Outcome(s):

- Successfully used LSTM for prediction of future weather of cities in Python.
- Visualizing results using Matplotlib.

BEYOND CURRICULUM

Experiment 9

Objective: Implement an Inception V3 for image classification.

Explanation:

InceptionV3 is a deep convolutional neural network designed for image classification, leveraging an advanced architecture with multiple-sized convolutional filters in parallel to capture multi-scale features. It employs factorized convolutions, asymmetric convolutions, and label smoothing to enhance efficiency and accuracy while reducing computational costs. The model, pre-trained on ImageNet, can be fine-tuned for custom datasets. In TensorFlow/Keras, InceptionV3 is implemented as a feature extractor or fine-tuned with additional layers. Its optimized design improves feature representation, making it effective for large-scale image classification tasks with high precision while maintaining computational efficiency.

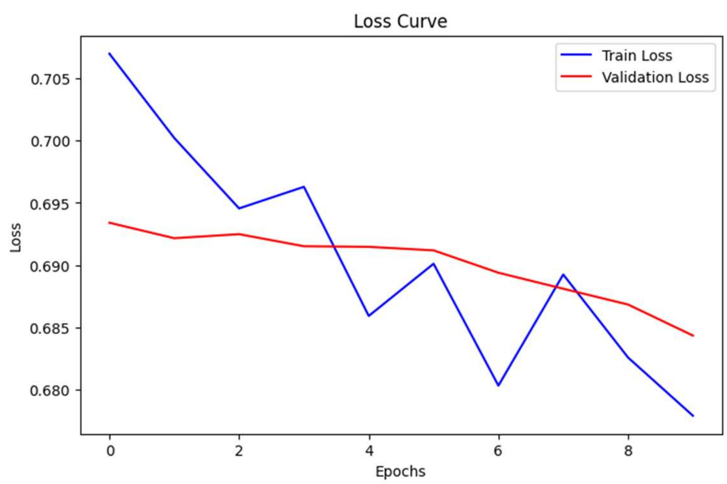
Code & Output:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.applications import InceptionV3

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten, Dropout, GlobalAveragePooling2D

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import fashion_mnist

from tensorflow.keras.utils import to_categorical

from tensorflow.keras.preprocessing.image import ImageDataGenerator


# Enable mixed precision training (if supported)

tf.keras.mixed_precision.set_global_policy('mixed_float16')
```

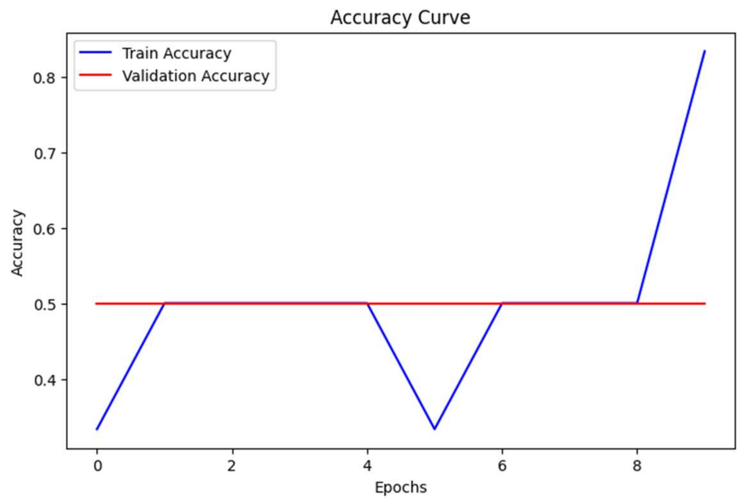
```
# Load Fashion MNIST dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Convert grayscale to RGB by expanding dimensions
x_train = np.repeat(x_train[..., np.newaxis], 3, -1)
x_test = np.repeat(x_test[..., np.newaxis], 3, -1)

# Normalize pixel values
x_train, x_test = x_train.astype('float32') / 255.0, x_test.astype('float32') / 255.0

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

# Create TensorFlow dataset with optimized pipeline
def preprocess(image, label):
    image = tf.image.resize(image, (150, 150))  # Adjusted for InceptionV3 input size
    return image, label

train_dataset = (tf.data.Dataset.from_tensor_slices((x_train, y_train))
                 .map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)
                 .cache()
                 .batch(32)
                 .prefetch(tf.data.AUTOTUNE))

test_dataset = (tf.data.Dataset.from_tensor_slices((x_test, y_test))
                .map(preprocess, num_parallel_calls=tf.data.AUTOTUNE)
```

```
        .cache()
        .batch(32)
        .prefetch(tf.data.AUTOTUNE))


# Load InceptionV3 model without top layer
base_model = InceptionV3(weights='imagenet', include_top=False, input_shape=(150,
150, 3))
for layer in base_model.layers:
    layer.trainable = False  # Freeze convolutional layers


# Add custom layers on top
x = GlobalAveragePooling2D()(base_model.output)
x = Dense(256, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(128, activation='relu')(x)
x = Dense(10, activation='softmax', dtype='float32')(x)  # Ensure correct dtype with mixed
precision


model = Model(inputs=base_model.input, outputs=x)
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])


# Train model
history = model.fit(train_dataset, epochs=5, validation_data=test_dataset)  # Reduced
epochs to 5


# Plot training and validation loss
```

```
plt.figure(figsize=(8, 5))
plt.plot(history.history['loss'], label='Train Loss', color='blue')
plt.plot(history.history['val_loss'], label='Validation Loss', color='red')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Curve')
plt.show()
```




```
# Plot training and validation accuracy
plt.figure(figsize=(8, 5))
plt.plot(history.history['accuracy'], label='Train Accuracy', color='blue')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Curve')
plt.show()
```



Learning Outcome(s):

- Successfully implemented an Inception V3 for image classification.
- Visualizing results using Matplotlib.

BEYOND CURRICULUM

Experiment 10

Objective: Implement Bi-directional LSTM for text classification.

Explanation:

A Bi-directional Long Short-Term Memory (Bi-LSTM) network enhances text classification by capturing both past and future dependencies in a sequence. Unlike standard LSTMs, which process text in one direction, Bi-LSTM consists of two LSTMs running in opposite directions, improving context understanding. This model is effective for sentiment analysis, spam detection, and document classification. Implemented in TensorFlow/Keras, it typically includes an embedding layer, Bi-LSTM layers, and dense layers for classification. Bi-LSTM improves accuracy by leveraging complete context, making it superior for processing long-range dependencies in natural language processing tasks.

Code & Output:

```
import numpy as np

import tensorflow as tf

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad_sequences

import matplotlib.pyplot as plt


# Sample dataset

texts = ["I love this product!", "This is the worst experience ever.", "Absolutely fantastic!", "Not good at all.", "Great service and fast delivery.", "Terrible quality, do not buy!", "Excellent and reliable.", "Would not recommend."]

labels = [1, 0, 1, 0, 1, 0, 1, 0]  # 1: Positive, 0: Negative
```

```
# Tokenization and padding
max_words = 10000
max_len = 20
tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
padded_sequences = pad_sequences(sequences, maxlen=max_len, padding='post')

# Convert labels to numpy array
labels = np.array(labels)

# Split dataset into training and validation sets
split = int(0.8 * len(texts))
x_train, x_val = padded_sequences[:split], padded_sequences[split:]
y_train, y_val = labels[:split], labels[split:]

# Define Bi-LSTM model
model = Sequential([
    Embedding(input_dim=max_words, output_dim=64, input_length=max_len),
    Bidirectional(LSTM(64, return_sequences=True)),
    Bidirectional(LSTM(32)),
    Dense(64, activation='relu'),
    Dropout(0.5),
    Dense(1, activation='sigmoid')
])
```

Compile model

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

Train model

history = model.fit(x_train, y_train, epochs=50, validation_data=(x_val, y_val), batch_size=2)

Plot training and validation loss

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

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

plt.plot(history.history['val_loss'], label='Validation Loss', color='red')

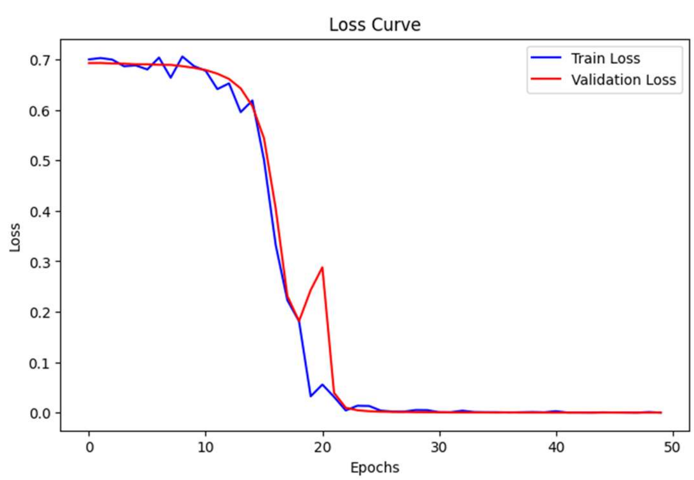
plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.title('Loss Curve')

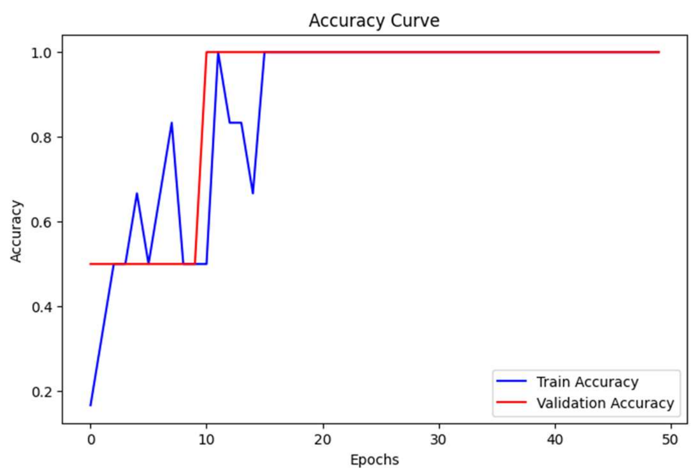
plt.show()



Plot training and validation accuracy

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

```
plt.plot(history.history['accuracy'], label='Train Accuracy', color='blue')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy', color='red')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Accuracy Curve')
plt.show()
```



Learning Outcome(s):

- Successfully implemented Bi-directional LSTM for text classification.
- Visualizing results using Matplotlib.