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.

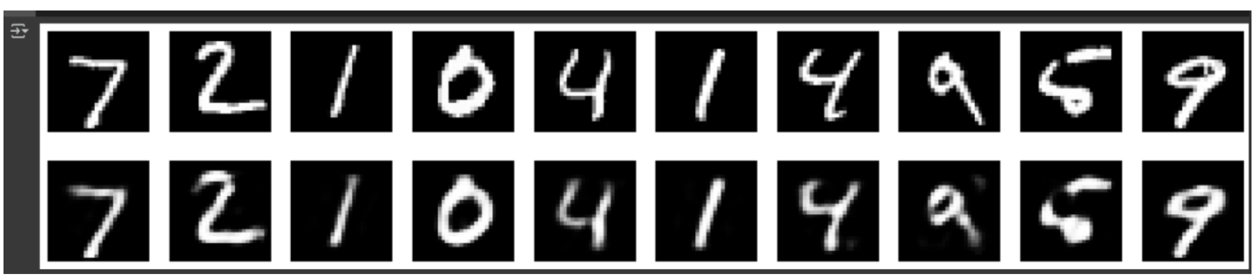
Code & Output:

```
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)
```

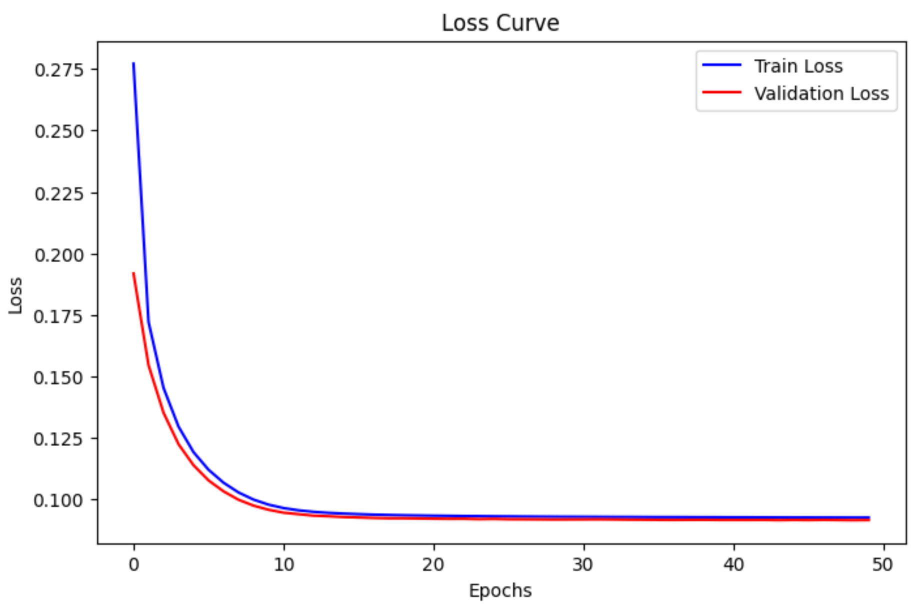
```
Epoch 1/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - loss: 0.3817 - val_loss: 0.1918
Epoch 2/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 10ms/step - loss: 0.1816 - val_loss: 0.1546
Epoch 3/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 2s 9ms/step - loss: 0.1500 - val_loss: 0.1352
Epoch 4/50
235/235 ━━━━━━━━━━━━━━━━━━━━ 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))
```

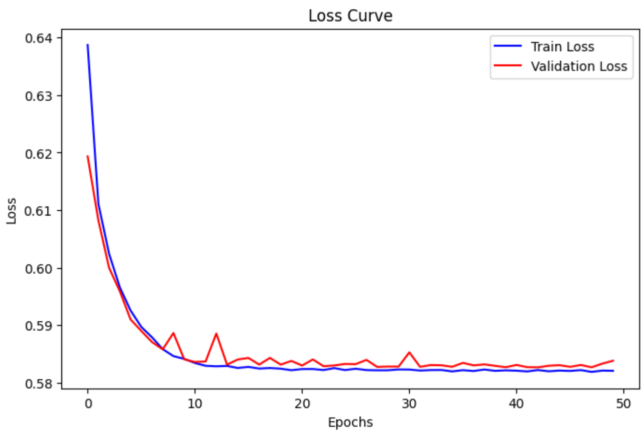
```
# Define encoding dimension
encoding_dim = 128  # Increased dimension for CIFAR-10

# 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)
```

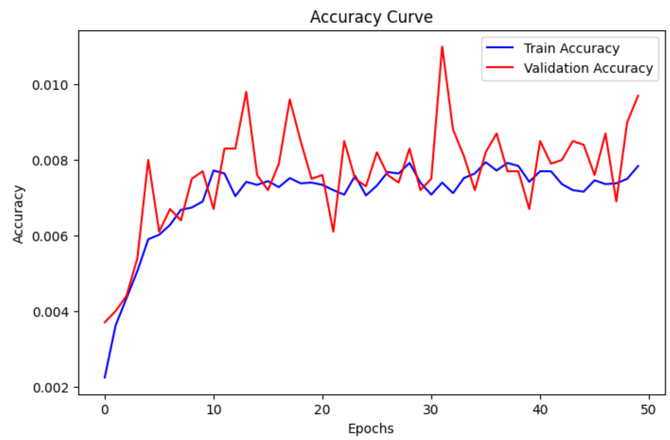
```
Epoch 1/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 13s 60ms/step - accuracy: 0.0017 - loss: 0.6593 - val_accuracy: 0.0037 - val_loss: 0.6193
Epoch 2/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 11s 55ms/step - accuracy: 0.0035 - loss: 0.6144 - val_accuracy: 0.0040 - val_loss: 0.6082
Epoch 3/50
196/196 ━━━━━━━━━━━━━━━━━━━━ 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.

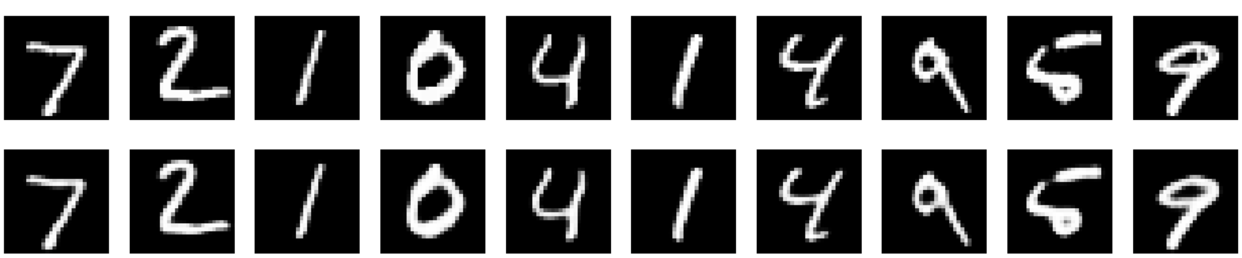
Code & Output:

```
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)
```

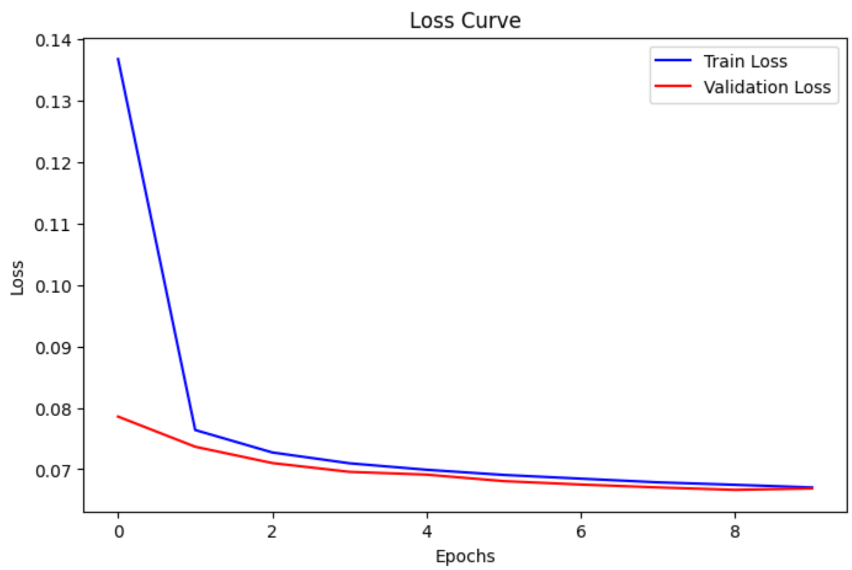
```
Epoch 1/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 169s 708ms/step - loss: 0.2436 - val_loss: 0.0786
Epoch 2/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 202s 708ms/step - loss: 0.0779 - val_loss: 0.0737
Epoch 3/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 199s 694ms/step - loss: 0.0734 - val_loss: 0.0710
Epoch 4/10
235/235 ━━━━━━━━━━━━━━━━━━━━ 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

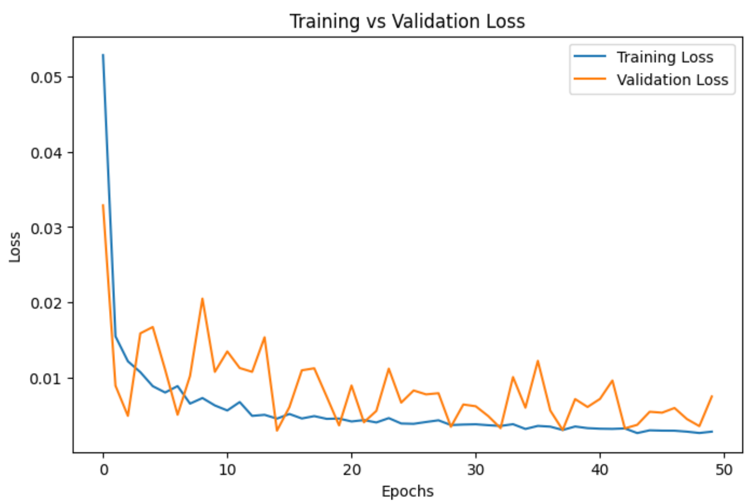
history = model.fit(X_train, y_train, epochs=50, batch_size=32, validation_data=(X_test, y_test))

```
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

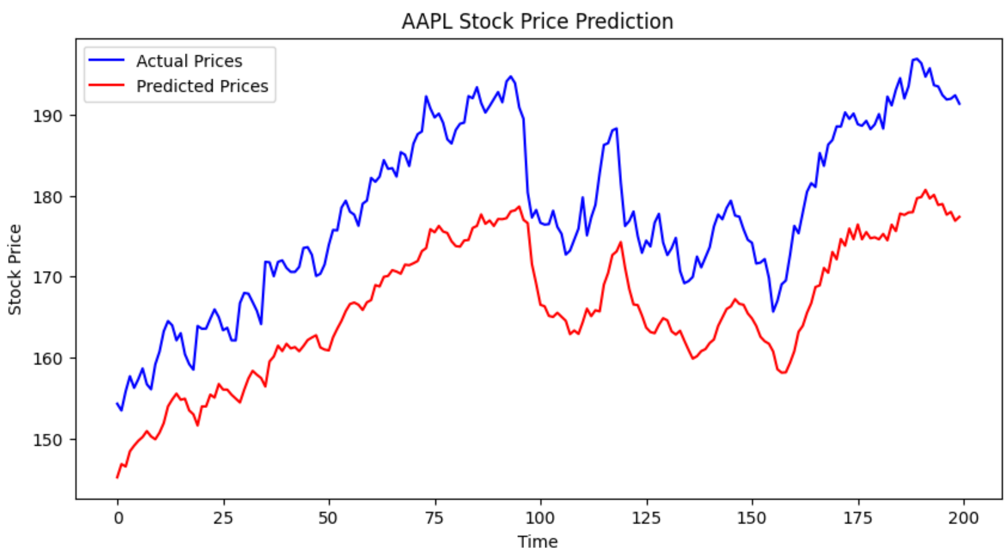
plt.figure(figsize=(8,5))

```
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:

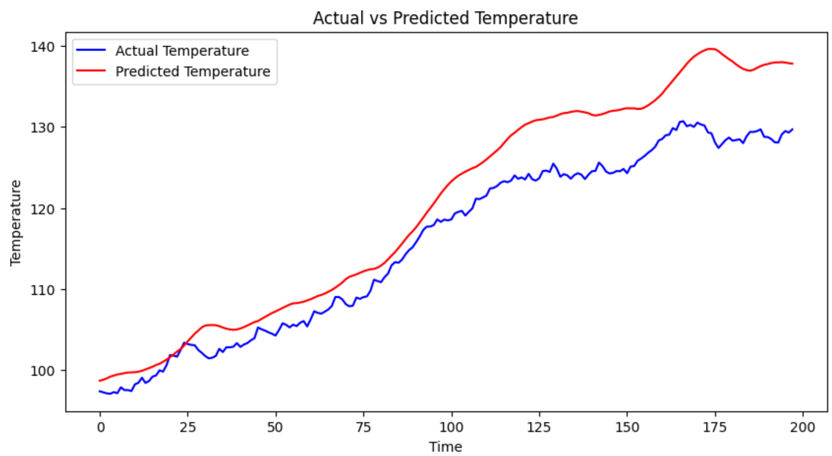
```
import numpy as np

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):
```

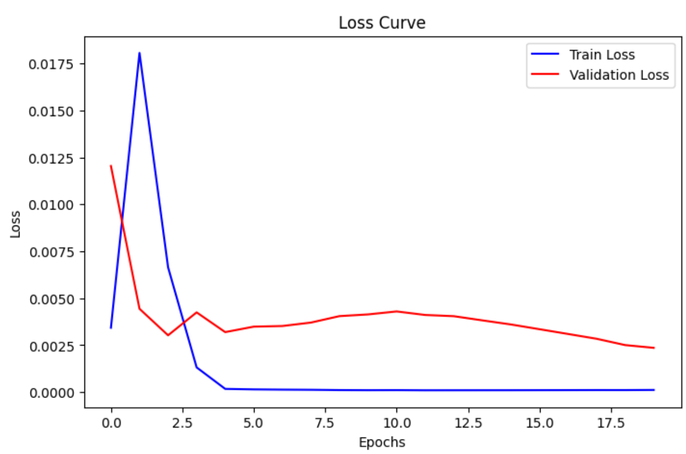
```
    X, y = [], []
    for i in range(len(data) - time_steps):
        X.append(data[i:i+time_steps])
        y.append(data[i+time_steps])
    return np.array(X), np.array(y)
scaler = MinMaxScaler()
temp_data_scaled = scaler.fit_transform(temp_data.reshape(-1, 1))
time_steps = 10
X, y = create_dataset(temp_data_scaled, 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.