lp5

May 6, 2025

EXP - 6

Linear regression by using Deep Neural network: Implement Boston housing price prediction problem by Linear regression using Deep Neural network. Use Boston House price prediction dataset

[11]:
```
import tensorflow as tf
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
import pandas as pd
```

[16]:
```
# import data

data = pd.read_csv(r'/Users/sakshisachinpotdar/Downloads/1_boston_housing.csv')
print(data.head())
```

```
      crim    zn  indus  chas    nox     rm   age     dis  rad  tax  ptratio  \
0  0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296     15.3
1  0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242     17.8
2  0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242     17.8
3  0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222     18.7
4  0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222     18.7

        b  lstat  MEDV
0  396.90   4.98  24.0
1  396.90   9.14  21.6
2  392.83   4.03  34.7
3  394.63   2.94  33.4
4  396.90   5.33  36.2
```

[18]:
```
# SPECIFY X & Y
# then train_test

y = data['MEDV']
x = data.drop(columns = ['MEDV'])
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2,␣
↪random_state = 32)
```

[21]:
```
# standardise fts

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

[25]:
```
# BUILD MODEL

model = Sequential()

model.add(Dense(64, input_dim = x_train.shape[1], activation = 'relu')) # 1st␣
↪hidden layer with 64 neurons
model.add(Dense(32, activation = 'relu' ))  # 2nd hidden layer with 32 neurons
model.add(Dense(1)) #Single output
```

[26]:
```
# MODEL COMPILATION
model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

[27]:
```
# TRAIN MODEL

model.fit(x_train, y_train, epochs = 100, batch_size = 32, validation_data =␣
↪(x_test, y_test))
```

```
Epoch 1/100
13/13                 0s 5ms/step - loss:
574.0211 - val_loss: 530.7908
Epoch 2/100
13/13                 0s 2ms/step - loss:
502.6110 - val_loss: 466.1854
Epoch 3/100
13/13                 0s 2ms/step - loss:
395.8426 - val_loss: 386.9705
Epoch 4/100
13/13                 0s 2ms/step - loss:
342.1078 - val_loss: 293.5982
Epoch 5/100
13/13                 0s 2ms/step - loss:
239.8614 - val_loss: 204.0109
Epoch 6/100
13/13                 0s 3ms/step - loss:
175.3199 - val_loss: 141.3174
Epoch 7/100
13/13                 0s 2ms/step - loss:
98.3913 - val_loss: 110.9292
Epoch 8/100
13/13                 0s 2ms/step - loss:
78.4389 - val_loss: 90.1927
```

```
Epoch 9/100
13/13            0s 2ms/step - loss:
57.6758 - val_loss: 72.1971
Epoch 10/100
13/13            0s 2ms/step - loss:
45.9713 - val_loss: 59.1600
Epoch 11/100
13/13            0s 2ms/step - loss:
34.1701 - val_loss: 50.5682
Epoch 12/100
13/13            0s 2ms/step - loss:
30.2187 - val_loss: 44.5006
Epoch 13/100
13/13            0s 2ms/step - loss:
26.8668 - val_loss: 40.2278
Epoch 14/100
13/13            0s 2ms/step - loss:
32.7378 - val_loss: 36.9662
Epoch 15/100
13/13            0s 2ms/step - loss:
23.5899 - val_loss: 34.6057
Epoch 16/100
13/13            0s 2ms/step - loss:
23.8061 - val_loss: 32.1485
Epoch 17/100
13/13            0s 2ms/step - loss:
23.0260 - val_loss: 30.5701
Epoch 18/100
13/13            0s 2ms/step - loss:
22.0493 - val_loss: 29.0889
Epoch 19/100
13/13            0s 2ms/step - loss:
21.7630 - val_loss: 27.9306
Epoch 20/100
13/13            0s 2ms/step - loss:
20.3039 - val_loss: 26.7396
Epoch 21/100
13/13            0s 2ms/step - loss:
15.8878 - val_loss: 25.7958
Epoch 22/100
13/13            0s 2ms/step - loss:
16.0272 - val_loss: 25.1617
Epoch 23/100
13/13            0s 2ms/step - loss:
19.3797 - val_loss: 24.3228
Epoch 24/100
13/13            0s 2ms/step - loss:
19.1960 - val_loss: 23.8775
```

```
Epoch 25/100
13/13              0s 2ms/step - loss:
16.1112 - val_loss: 23.0935
Epoch 26/100
13/13              0s 2ms/step - loss:
14.5181 - val_loss: 22.7338
Epoch 27/100
13/13              0s 2ms/step - loss:
16.4957 - val_loss: 21.9592
Epoch 28/100
13/13              0s 2ms/step - loss:
13.3504 - val_loss: 21.9457
Epoch 29/100
13/13              0s 2ms/step - loss:
13.3903 - val_loss: 21.3593
Epoch 30/100
13/13              0s 2ms/step - loss:
12.0795 - val_loss: 21.2926
Epoch 31/100
13/13              0s 4ms/step - loss:
13.2743 - val_loss: 20.7340
Epoch 32/100
13/13              0s 2ms/step - loss:
13.1588 - val_loss: 20.6477
Epoch 33/100
13/13              0s 2ms/step - loss:
11.8908 - val_loss: 20.3710
Epoch 34/100
13/13              0s 2ms/step - loss:
13.2728 - val_loss: 20.0850
Epoch 35/100
13/13              0s 2ms/step - loss:
13.6316 - val_loss: 20.0262
Epoch 36/100
13/13              0s 2ms/step - loss:
16.2454 - val_loss: 19.7193
Epoch 37/100
13/13              0s 2ms/step - loss:
12.4100 - val_loss: 19.5555
Epoch 38/100
13/13              0s 2ms/step - loss:
12.8917 - val_loss: 19.6123
Epoch 39/100
13/13              0s 2ms/step - loss:
13.6737 - val_loss: 19.4560
Epoch 40/100
13/13              0s 2ms/step - loss:
11.5287 - val_loss: 19.3714
```

```
Epoch 41/100
13/13            0s 2ms/step - loss:
13.4470 - val_loss: 18.9986
Epoch 42/100
13/13            0s 2ms/step - loss:
9.5816 - val_loss: 19.1964
Epoch 43/100
13/13            0s 2ms/step - loss:
9.8440 - val_loss: 18.9520
Epoch 44/100
13/13            0s 4ms/step - loss:
9.7949 - val_loss: 18.9620
Epoch 45/100
13/13            0s 2ms/step - loss:
12.3425 - val_loss: 18.8336
Epoch 46/100
13/13            0s 2ms/step - loss:
10.8069 - val_loss: 18.8129
Epoch 47/100
13/13            0s 2ms/step - loss:
10.6927 - val_loss: 18.6624
Epoch 48/100
13/13            0s 2ms/step - loss:
9.7675 - val_loss: 18.6233
Epoch 49/100
13/13            0s 2ms/step - loss:
11.4991 - val_loss: 18.6640
Epoch 50/100
13/13            0s 2ms/step - loss:
9.4074 - val_loss: 18.5716
Epoch 51/100
13/13            0s 2ms/step - loss:
8.8536 - val_loss: 18.5260
Epoch 52/100
13/13            0s 2ms/step - loss:
9.8002 - val_loss: 18.4020
Epoch 53/100
13/13            0s 2ms/step - loss:
12.8017 - val_loss: 18.3972
Epoch 54/100
13/13            0s 2ms/step - loss:
12.3081 - val_loss: 18.5454
Epoch 55/100
13/13            0s 2ms/step - loss:
8.5845 - val_loss: 18.3113
Epoch 56/100
13/13            0s 2ms/step - loss:
10.1593 - val_loss: 18.1546
```

```
Epoch 57/100
13/13            0s 3ms/step - loss:
9.5331 - val_loss: 18.2164
Epoch 58/100
13/13            0s 2ms/step - loss:
9.9207 - val_loss: 18.2329
Epoch 59/100
13/13            0s 2ms/step - loss:
9.7162 - val_loss: 18.1948
Epoch 60/100
13/13            0s 2ms/step - loss:
9.6262 - val_loss: 18.2148
Epoch 61/100
13/13            0s 2ms/step - loss:
10.6908 - val_loss: 18.2444
Epoch 62/100
13/13            0s 2ms/step - loss:
11.5055 - val_loss: 18.0579
Epoch 63/100
13/13            0s 2ms/step - loss:
9.5257 - val_loss: 18.1005
Epoch 64/100
13/13            0s 3ms/step - loss:
12.5958 - val_loss: 18.0821
Epoch 65/100
13/13            0s 2ms/step - loss:
8.7596 - val_loss: 18.0166
Epoch 66/100
13/13            0s 2ms/step - loss:
8.2777 - val_loss: 18.0720
Epoch 67/100
13/13            0s 2ms/step - loss:
9.3511 - val_loss: 18.0093
Epoch 68/100
13/13            0s 2ms/step - loss:
7.5161 - val_loss: 17.9276
Epoch 69/100
13/13            0s 2ms/step - loss:
7.9653 - val_loss: 17.7819
Epoch 70/100
13/13            0s 2ms/step - loss:
7.5801 - val_loss: 17.7847
Epoch 71/100
13/13            0s 2ms/step - loss:
10.7447 - val_loss: 17.8071
Epoch 72/100
13/13            0s 2ms/step - loss:
7.9200 - val_loss: 17.8893
```

```
Epoch 73/100
13/13            0s 2ms/step - loss:
10.2547 - val_loss: 17.9021
Epoch 74/100
13/13            0s 3ms/step - loss:
8.1569 - val_loss: 17.8021
Epoch 75/100
13/13            0s 2ms/step - loss:
7.7583 - val_loss: 17.7611
Epoch 76/100
13/13            0s 2ms/step - loss:
10.1824 - val_loss: 17.6027
Epoch 77/100
13/13            0s 2ms/step - loss:
9.4185 - val_loss: 17.7168
Epoch 78/100
13/13            0s 2ms/step - loss:
9.4211 - val_loss: 17.8408
Epoch 79/100
13/13            0s 2ms/step - loss:
9.3143 - val_loss: 17.6500
Epoch 80/100
13/13            0s 2ms/step - loss:
8.2847 - val_loss: 17.7216
Epoch 81/100
13/13            0s 2ms/step - loss:
8.8211 - val_loss: 17.7020
Epoch 82/100
13/13            0s 2ms/step - loss:
8.5291 - val_loss: 17.7535
Epoch 83/100
13/13            0s 3ms/step - loss:
8.6597 - val_loss: 17.6259
Epoch 84/100
13/13            0s 2ms/step - loss:
7.6142 - val_loss: 17.6598
Epoch 85/100
13/13            0s 2ms/step - loss:
6.6553 - val_loss: 17.8108
Epoch 86/100
13/13            0s 2ms/step - loss:
7.4534 - val_loss: 17.5466
Epoch 87/100
13/13            0s 2ms/step - loss:
9.1237 - val_loss: 17.6228
Epoch 88/100
13/13            0s 2ms/step - loss:
7.3712 - val_loss: 17.5148
```

```
Epoch 89/100
13/13              0s 2ms/step - loss:
7.7251 - val_loss: 17.6035
Epoch 90/100
13/13              0s 2ms/step - loss:
7.8541 - val_loss: 17.5903
Epoch 91/100
13/13              0s 3ms/step - loss:
7.8447 - val_loss: 17.4894
Epoch 92/100
13/13              0s 2ms/step - loss:
10.6556 - val_loss: 17.4689
Epoch 93/100
13/13              0s 2ms/step - loss:
8.1113 - val_loss: 17.5697
Epoch 94/100
13/13              0s 2ms/step - loss:
9.6750 - val_loss: 17.4576
Epoch 95/100
13/13              0s 2ms/step - loss:
8.1899 - val_loss: 17.6628
Epoch 96/100
13/13              0s 2ms/step - loss:
8.8456 - val_loss: 17.3911
Epoch 97/100
13/13              0s 2ms/step - loss:
8.1951 - val_loss: 17.5429
Epoch 98/100
13/13              0s 2ms/step - loss:
7.6914 - val_loss: 17.4342
Epoch 99/100
13/13              0s 3ms/step - loss:
7.9066 - val_loss: 17.3363
Epoch 100/100
13/13              0s 2ms/step - loss:
6.7606 - val_loss: 17.4322
```

[27]: <keras.src.callbacks.history.History at 0x13f726570>

[29]:
```
# Evaluate model

loss = model.evaluate(x_test, y_test)
print(f'Model loss (MSE): {loss}')
```

```
4/4              0s 3ms/step - loss:
19.1693
Model loss (MSE): 17.432201385498047
```

[31]:
```python
# make predictions

y_pred = model.predict(x_test)
print("Predictions: ", y_pred[:5])  # Print the first 5 predictions
```

```
4/4           0s 6ms/step
Predictions:  [[13.574309]
 [28.991796]
 [15.981   ]
 [20.510895]
 [24.87274 ]]
```

[ ]:
```python

```

Experiment 7

[2]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, LSTM, SpatialDropout1D
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import pandas as pd

# Load your CSV dataset
```

[3]:
```python
df = pd.read_csv('/Users/sakshisachinpotdar/Downloads/IMDB Dataset - IMDB␣
 ↪Dataset.csv')
```

[4]:
```python
# Let's assume your CSV has two columns: 'review' and 'sentiment'
reviews = df['review'].values
labels = df['sentiment'].values
```

[5]:
```python
# Convert string labels ('positive' and 'negative') to numerical labels (1 and␣
 ↪0)
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(labels)  # Converts 'positive' to 1 and␣
 ↪'negative' to 0
```

[6]:
```python
# Tokenize text
tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(reviews)
X = tokenizer.texts_to_sequences(reviews)
X = pad_sequences(X, padding='post', maxlen=100)  # Ensure same length
```

[7]:
```
# Split into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
 ↪random_state=42)
```

[10]:
```
model = Sequential()
model.add(Embedding(input_dim=10000, output_dim=128))
model.add(SpatialDropout1D(0.2))  # Dropout to prevent overfitting
model.add(LSTM(100, dropout=0.2, recurrent_dropout=0.2))
model.add(Dense(1, activation='sigmoid'))
```

[11]:
```
model.compile(loss='binary_crossentropy', optimizer=Adam(learning_rate=0.001),
 ↪metrics=['accuracy'])
```

[12]:
```
history = model.fit(X_train, y_train, epochs=5, batch_size=64,
 ↪validation_data=(X_test, y_test), verbose=2)
```

```
Epoch 1/5
625/625 - 65s - 104ms/step - accuracy: 0.7822 - loss: 0.4666 - val_accuracy:
0.8347 - val_loss: 0.3775
Epoch 2/5
625/625 - 64s - 102ms/step - accuracy: 0.8614 - loss: 0.3345 - val_accuracy:
0.8549 - val_loss: 0.3418
Epoch 3/5
625/625 - 66s - 106ms/step - accuracy: 0.8881 - loss: 0.2798 - val_accuracy:
0.8665 - val_loss: 0.3202
Epoch 4/5
625/625 - 65s - 104ms/step - accuracy: 0.9037 - loss: 0.2436 - val_accuracy:
0.8694 - val_loss: 0.3383
Epoch 5/5
625/625 - 65s - 104ms/step - accuracy: 0.9182 - loss: 0.2133 - val_accuracy:
0.8706 - val_loss: 0.3314
```

[13]:
```
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {accuracy}")
```

```
313/313              5s 16ms/step -
accuracy: 0.8744 - loss: 0.3240
Test Accuracy: 0.8705999851226807
```

[ ]:
```

```

EXPERIMENT 8

Convolutional neural network (CNN) Use MNIST Fashion Dataset and create a classifier to classify fashion clothing into categories.

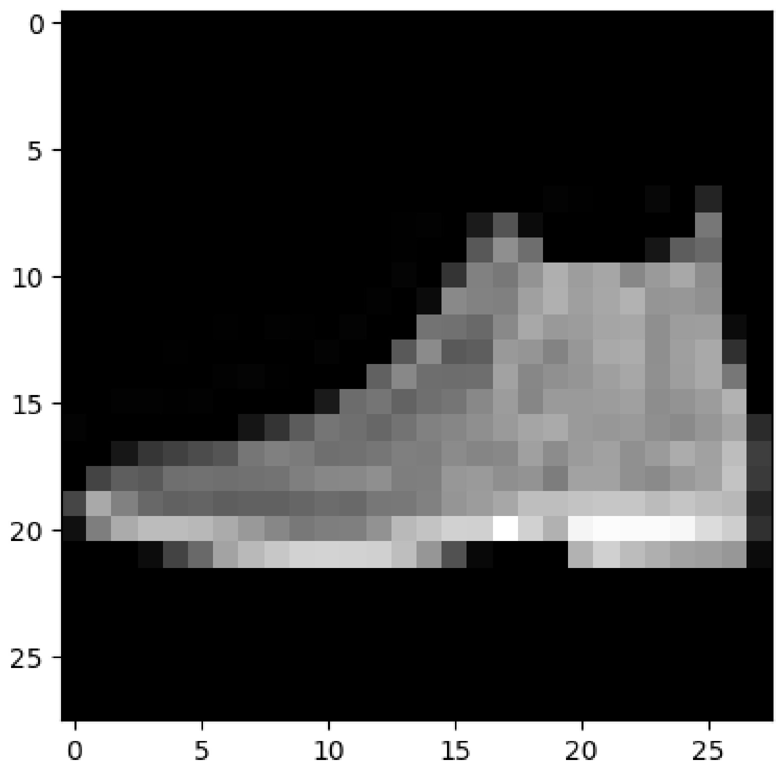
[19]:
```
import numpy as np
import matplotlib.pyplot as plt
```

```
from tensorflow.keras.datasets import fashion_mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, MaxPooling2D, Conv2D

# Load the Fashion MNIST dataset
(train_x, train_y), (test_x, test_y) = fashion_mnist.load_data()

# Normalize the data (important for good performance)
train_x = train_x.astype('float32') / 255.0
test_x = test_x.astype('float32') / 255.0

# Reshape the data to include the channel dimension (grayscale image)
train_x = train_x.reshape(-1, 28, 28, 1)
test_x = test_x.reshape(-1, 28, 28, 1)

# Define the model
model = Sequential()

# Add a Conv2D layer with 64 filters and ReLU activation function
model.add(Conv2D(filters=64, kernel_size=(3, 3), activation='relu',␣
↪input_shape=(28, 28, 1)))

# Add a MaxPooling2D layer to downsample the feature map
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the output to feed into a fully connected layer
model.add(Flatten())

# Add a Dense layer with 128 neurons and ReLU activation
model.add(Dense(128, activation='relu'))

# Add the output layer with 10 neurons (one for each class) and softmax␣
↪activation
model.add(Dense(10, activation='softmax'))

# Print the model summary
model.summary()

# Compile the model with the Adam optimizer and sparse categorical␣
↪cross-entropy loss
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',␣
↪metrics=['accuracy'])

# Train the model with a validation split of 0.2
model.fit(train_x, train_y, epochs=5, validation_split=0.2)

# Evaluate the model on the test data
```

```
loss, acc = model.evaluate(test_x, test_y)
print(f"Test loss: {loss}, Test accuracy: {acc}")

# Define the label names corresponding to the 10 classes
labels = ['t-shirt', 'trouser', 'pullover', 'dress', 'coat', 'sandal', 'shirt',␣
 ↪'sneaker', 'bag', 'ankle_boots']

# Make predictions on the first test image
predictions = model.predict(test_x[:1])

# Map the predicted label to the correct class
label = labels[np.argmax(predictions)]

# Plot the first test image
print("Predicted label:", label)
plt.imshow(test_x[:1][0], cmap='gray')  # Use 'gray' to show the image in␣
 ↪grayscale
plt.show()
```

```
Model: "sequential_1"


 Layer (type)                    Output Shape                  Param #

 conv2d_11 (Conv2D)              (None, 26, 26, 64)                640

 max_pooling2d_2 (MaxPooling2D)  (None, 13, 13, 64)                  0

 flatten_1 (Flatten)             (None, 10816)                       0

 dense (Dense)                   (None, 128)                 1,384,576

 dense_1 (Dense)                 (None, 10)                      1,290


 Total params: 1,386,506 (5.29 MB)


 Trainable params: 1,386,506 (5.29 MB)


 Non-trainable params: 0 (0.00 B)

Epoch 1/5
1500/1500            11s 7ms/step -
accuracy: 0.8120 - loss: 0.5336 - val_accuracy: 0.8932 - val_loss: 0.2919
Epoch 2/5
```

```
1500/1500             11s 8ms/step -
accuracy: 0.9040 - loss: 0.2668 - val_accuracy: 0.9057 - val_loss: 0.2605
Epoch 3/5
1500/1500             11s 8ms/step -
accuracy: 0.9191 - loss: 0.2192 - val_accuracy: 0.9082 - val_loss: 0.2455
Epoch 4/5
1500/1500             11s 7ms/step -
accuracy: 0.9350 - loss: 0.1760 - val_accuracy: 0.9154 - val_loss: 0.2403
Epoch 5/5
1500/1500             11s 7ms/step -
accuracy: 0.9478 - loss: 0.1440 - val_accuracy: 0.9106 - val_loss: 0.2651
313/313             0s 1ms/step -
accuracy: 0.9034 - loss: 0.2861
Test loss: 0.2759930491447449, Test accuracy: 0.9075000286102295
1/1             0s 21ms/step
Predicted label: ankle_boots
```



[ ]:

EXPERIMENT 9

Recurrent neural network (RNN) Use the Google stock prices dataset and design a time seriesanalysis and prediction system using RNN.

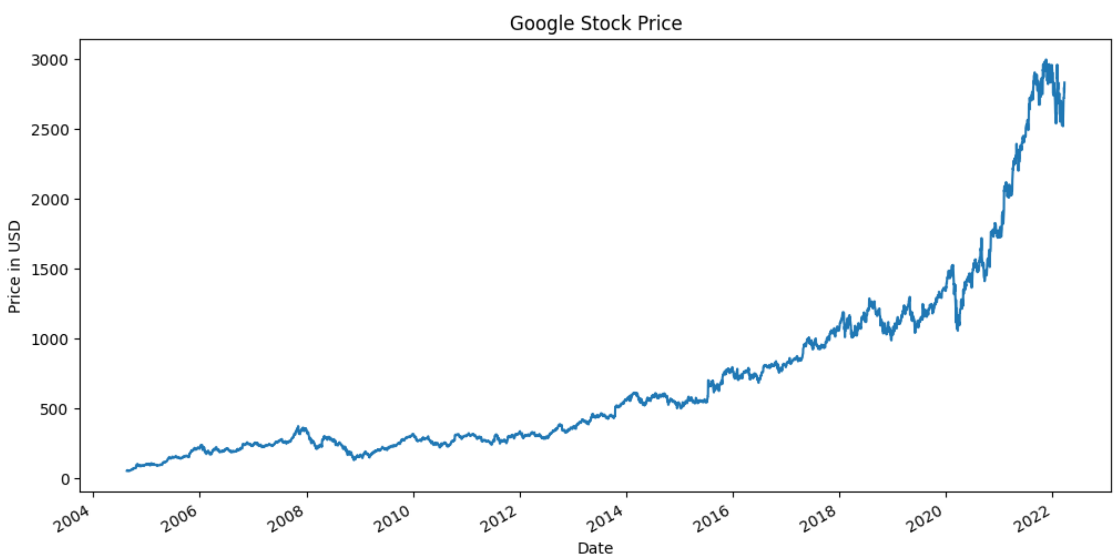
[15]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
```

[16]:
```
df = pd.read_csv("/Users/sakshisachinpotdar/Downloads/GOOGLE Stock Data set -␣
↪GOOGLE Stock Data set.csv")
```

[17]:
```
df = df[['Date', 'Close']]
```

[18]:
```
df['Date'] = pd.to_datetime(df['Date'])
df.set_index('Date', inplace=True)
```

[19]:
```
df['Close'].plot(figsize=(12, 6))
plt.title("Google Stock Price")
plt.ylabel("Price in USD")
plt.show()
```



[20]:
```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(df['Close'].values.reshape(-1, 1))
```

[21]:
```
look_back = 60  # Use the past 60 days to predict the next day's price
generator = TimeseriesGenerator(scaled_data, scaled_data, length=look_back,␣
↪batch_size=32)
```

[22]:
```
train_size = int(len(df) * 0.8)
train_data, test_data = scaled_data[:train_size], scaled_data[train_size:]
```

[23]:
```
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(look_back, 1)))
model.add(Dense(1))  # Output layer (single value for next price)
model.compile(optimizer='adam', loss='mean_squared_error')
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential models,
prefer using an `Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

[24]:
```
model.fit(generator, epochs=10, verbose=1)

# Make predictions
```

```
Epoch 1/10
```

```
/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
137/137           1s 6ms/step -
loss: 0.0839
Epoch 2/10
137/137           1s 6ms/step -
loss: 0.0189
Epoch 3/10
137/137           1s 7ms/step -
loss: 0.0088
Epoch 4/10
137/137           1s 7ms/step -
loss: 9.0410e-04
Epoch 5/10
137/137           1s 6ms/step -
loss: 1.9288e-04
Epoch 6/10
137/137           1s 7ms/step -
loss: 9.6494e-05
Epoch 7/10
137/137           1s 7ms/step -
loss: 8.1387e-05
Epoch 8/10
```

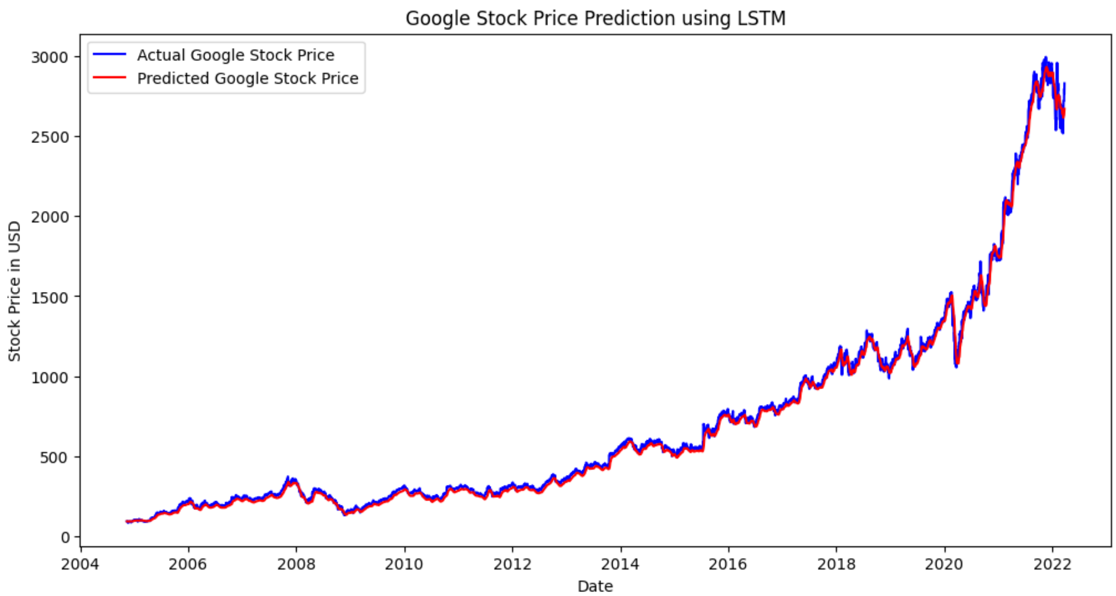
```
137/137          1s 7ms/step -
loss: 1.3209e-04
Epoch 9/10
137/137          1s 7ms/step -
loss: 8.2351e-05
Epoch 10/10
137/137          1s 6ms/step -
loss: 9.2830e-05
```

[24]: <keras.src.callbacks.history.History at 0x159c92810>

[25]:
```
predicted_stock_price = model.predict(generator)
```

```
137/137          0s 2ms/step
```

[26]:
```
# Inverse transform the predictions back to original scale
predicted_stock_price = scaler.inverse_transform(predicted_stock_price)

# Plot actual vs predicted stock prices
plt.figure(figsize=(12, 6))
plt.plot(df.index[look_back:], scaler.inverse_transform(scaled_data[look_back:
↪]), color='blue', label='Actual Google Stock Price')
plt.plot(df.index[look_back:], predicted_stock_price, color='red',␣
↪label='Predicted Google Stock Price')
plt.title("Google Stock Price Prediction using LSTM")
plt.xlabel("Date")
plt.ylabel("Stock Price in USD")
plt.legend()
plt.show()
```



```
[ ]:
```

```
[ ]:
```

```
[ ]:
```