ChatGPT

LSTM Stock Price Prediction in Python: Step-by-Step Explanation

The code uses Python libraries like **pandas** for data handling, **NumPy** for numerical operations, **scikit-learn** for data scaling, **Matplotlib** for plotting, and **Keras/TensorFlow** for building the neural network. Each part of the code does a specific job, explained below in simple terms.

Importing Libraries

The first step is to import needed Python packages. For example:

```
import pandas as pd      # pandas to load and handle data in tables (DataFrames)
import numpy as np       # NumPy for numerical operations on arrays
import matplotlib.pyplot as plt  # matplotlib for plotting graphs
from sklearn.preprocessing import MinMaxScaler  # for scaling data to [0,1]
from tensorflow.keras.models import Sequential  # high-level API to build models
from tensorflow.keras.layers import LSTM, Dropout, Dense  # neural network layers
```

- **pandas (`pd`)**: Simplifies reading data (like CSV files) and manipulating tables.
- **numpy (`np`)**: Provides fast array and math operations (like slicing and reshaping data).
- **matplotlib.pyplot (`plt`)**: Used to draw charts (e.g., lines showing stock prices over time).
- **MinMaxScaler**: A tool from scikit-learn that **scales numerical data** into a fixed range (usually 0 to 1) [1]. This helps different features have similar scale.
- **tensorflow.keras** layers (LSTM, Dropout, Dense): Used to build and train the neural network. Keras is a high-level library (using TensorFlow) that makes it easier to define models layer by layer.

Loading the Data

Next, the code loads the historical Google stock prices (e.g., from a CSV file):

```
df = pd.read_csv('GOOG.csv')        # read data into a pandas DataFrame
df.head()                           # show first few rows (date, open, high, low, close, etc.)
```

This reads a CSV file (such as one downloaded from Yahoo Finance) into a table called `df`. Each row is a day of stock prices (with columns like `Date`, `Open`, `Close`, etc.). The code often focuses on the **closing price** (or both open/close) as the value to predict.

- We might convert the date column to Python datetime and set it as index (so plots on the time axis make sense).

- For example:

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

(No external source needed for these basic pandas steps.)

Preprocessing and Scaling Data

Neural networks work best when input data is on a similar scale (usually 0 to 1). The code uses **MinMaxScaler** for this. Here's how it works:

```
data = df['Close'].values.reshape(-1, 1)  # get closing prices as a 2D array
scaler = MinMaxScaler(feature_range=(0, 1))
scaled_data = scaler.fit_transform(data)  # scale values to the 0–1 range
```

- `MinMaxScaler` **linearly transforms** each value so that the minimum becomes 0 and the maximum becomes 1 [1]. For example, a price of 100 might become 0.23 if 100 was 23% of the way between the min and max in the data.
- This ensures **all inputs have the same range**, preventing one large-valued feature from dominating learning [2] [1]. In stock data, raw prices can vary widely (hundreds or thousands), so scaling them helps the neural network train more effectively.
- We use `fit_transform` on the training data to learn the scaling, and later will use the same scaler to transform the test data and undo (inverse) the scaling on predictions.

Creating Training and Test Sets

We split the data into a “training” set (used to train the model) and a “test” set (to check predictions). A common split is 80% train, 20% test. For example:

```
train_size = int(len(scaled_data) * 0.8)
train_data = scaled_data[:train_size]   # first 80% of data for training
test_data = scaled_data[train_size:]    # last 20% for testing
```

- **Why split data?** We want to train on past prices and test on unseen future prices to evaluate performance.
- The split is done *in time order*, not randomly, because stock data is sequential.

Next, we **create input-output sequences** for the LSTM model. LSTMs learn from windows of past time steps. A common approach is to use the past 60 days to predict the next day. For example:

```
X_train, y_train = [], []
for i in range(60, len(train_data)):
    X_train.append(train_data[i-60:i, 0])  # past 60 days' prices
    y_train.append(train_data[i, 0])      # current day's price
X_train, y_train = np.array(X_train), np.array(y_train)
```

- Here, `X_train` is a list of arrays: each array has 60 days of the scaled prices, and `y_train` is the price on the following day.
- After the loop, we convert `X_train` and `y_train` to NumPy arrays for efficiency.

Before feeding `X_train` to the LSTM, we **reshape** it into a 3D array: `(samples, timesteps, features)` as required by Keras [3]. In our case, there is 1 feature (the price):

```
X_train = X_train.reshape((X_train.shape[0], X_train.shape[1], 1))
```

- This tells the LSTM: "I have *N* samples, each with 60 time steps, and 1 feature per step." Keras LSTM layers expect input shape **(batch_size, time_steps, features)** [3]. If there were more features (e.g. volume or technical indicators), the last dimension would be larger.

Building the LSTM Model

Now we construct the neural network. A typical LSTM model for time-series prediction might look like:

```
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(60, 1)))
model.add(Dropout(0.2))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dropout(0.2))
model.add(Dense(units=25))
model.add(Dense(units=1))
```

Let's break this down **layer by layer**:

- `Sequential()`: Starts a linear stack of layers.
- `LSTM(units=50, return_sequences=True, input_shape=(60,1))`:
- This is the first LSTM layer with 50 "units" (neurons).
- `input_shape=(60,1)` tells it to expect sequences of 60 steps with 1 feature each (the 60-day price windows we built).
- `return_sequences=True` means this LSTM returns the full sequence of hidden states (one for each of the 60 time steps) [4]. This is needed because we add another LSTM layer after it. (If we only had one LSTM layer, we could set this to False and it would output just the final time step.)
- `Dropout(0.2)`: This layer randomly "drops" 20% of the LSTM's outputs each training step [5]. It helps **prevent overfitting** by ensuring the model does not become too reliant on any one neuron's output [5] [6]. Over many iterations, dropout forces the network to learn redundant representations, which generally improves generalization to new data.

- `LSTM(units=50, return_sequences=False)`: A second LSTM layer with 50 units. `return_sequences=False` (the default) means it will output only the hidden state after the last time step (not the full sequence). This single vector is fed into the next layers.
- `Dropout(0.2)`: Another dropout layer after the second LSTM for the same reason (regularization).
- `Dense(units=25)`: A fully-connected ("Dense") layer with 25 neurons. This layer combines the LSTM outputs into higher-level features. It helps in mapping the 50-dimensional LSTM output to the final prediction dimension.
- `Dense(units=1)`: The final output layer with 1 neuron, giving the **predicted price** (in scaled form).

We then compile the model:

```
model.compile(optimizer='adam', loss='mean_squared_error')
```

- **Optimizer:** `'adam'` is a popular choice (an adaptive gradient method) that often works well. It adjusts learning rates automatically for efficient training.
- **Loss function:** `'mean_squared_error'` (MSE) measures the average squared difference between predicted and actual prices. It's common for regression tasks like price prediction.

(Citations: Dropout explanation [5], LSTM I/O shapes [3] [4], dropout prevents overfitting [6].)

Training the Model

With the model defined, we train it on our data:

```
history = model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=1)
```

- **epochs=100** means the model sees the entire training set 100 times, updating its weights each time. More epochs allow the model to learn more, but too many can cause overfitting.
- **batch_size=32** means it processes 32 samples at a time before updating weights.
- During training, the model uses **backpropagation through time** to adjust its internal parameters (weights) to minimize the MSE loss on the training data [7].

After training, we have a model tuned to the patterns in the historical data.

Making Predictions

We now use the trained model to predict future prices on the test set. First, we create the input sequences from the last part of the data (just like for training). Suppose `X_test` and `y_test` are prepared similarly:

```
X_test, y_test = [], []
for i in range(60, len(test_data)):
    X_test.append(test_data[i-60:i, 0])
    y_test.append(test_data[i, 0])
```

```
X_test = np.array(X_test)
X_test = X_test.reshape((X_test.shape[0], X_test.shape[1], 1))
```

Now we predict:

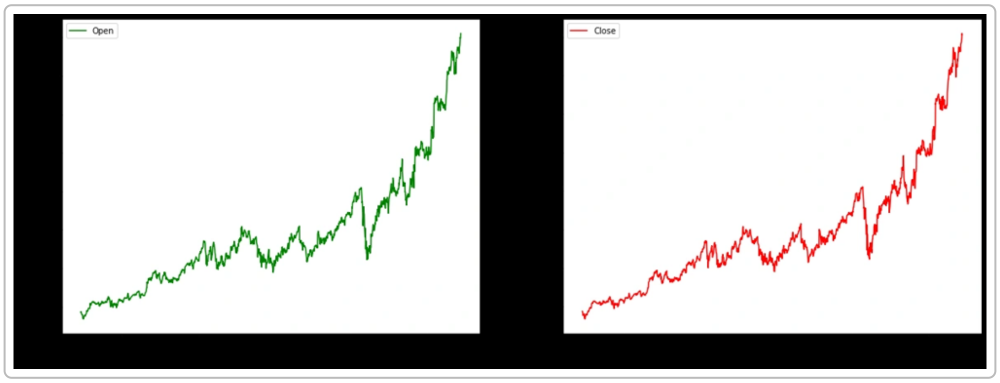
```
predictions = model.predict(X_test)         # model outputs scaled prices
predictions = scaler.inverse_transform(predictions)  # convert back to original price scale
```

- `model.predict(X_test)` returns scaled values between 0 and 1. We use `scaler.inverse_transform` to convert those scaled predictions back to actual price values in dollars.
- This gives us the predicted closing prices corresponding to each date in the test period.

We also invert-scale the `y_test` data similarly, to compare in actual prices.

Visualizing the Results

To see how well the model did, we plot the **actual** vs. **predicted** prices:



Visualization: Historical Google stock prices (green = "Open", red = "Close") over time. Color-coding helps distinguish the two price series 8.

```
plt.figure(figsize=(10,6))
plt.plot(y_test, color='blue', label='Actual Prices')
plt.plot(predictions, color='red', label='Predicted Prices')
plt.title('Actual vs. Predicted Google Stock Prices')
plt.xlabel('Time Steps')
plt.ylabel('Price (USD)')
plt.legend()
plt.show()
```

- In the plot above, **blue** would be the true prices from the data (after scaling back to actual dollars) and **red** would be the model's predicted prices for the same dates.

- A close alignment (red line following blue line) would indicate good performance. (Note: In practice, you should plot enough points to see the trend. We must check that the arrays `y_test` and `predictions` line up by date.)

The example image (green/red lines) comes from plotting open and close prices [8]. In our case, you would label actual vs. predicted differently. But the main idea is that **visualizing predictions helps us quickly see if the model is capturing the stock's movements**.

Potential Improvements and Next Steps

This basic LSTM model can be improved or extended in many ways:

- **Tune hyperparameters:** Change network size (more or fewer LSTM units), number of layers, dropout rates, learning rate, batch size, or number of epochs to find a better setting. Methods like grid search or Keras Tuner can help.
- **Use additional data features:** Instead of using only closing price, include other relevant inputs (features) such as open/high/low prices, trading volume, or technical indicators (moving averages, RSI, etc.). These can provide more information for the model to learn from. Many stock-prediction projects incorporate indicators to boost accuracy.
- **Add more LSTM layers or sequence steps:** You might experiment with deeper networks (more stacked LSTM layers) or using longer history windows (more than 60 days), especially if you have a lot of data.
- **Cross-validation:** Since financial data is time-series, you can try walk-forward validation or expanding-window methods to better estimate performance, rather than a single train/test split.
- **Use advanced architectures:** Explore variants like GRUs or bidirectional LSTMs, or even attention-based models (Transformers) for time-series.
- **Evaluation metrics:** Besides plotting, compute numerical metrics like **Root Mean Squared Error (RMSE)** or **Mean Absolute Error (MAE)** on the test set to quantify accuracy [9]. Lower error means better predictions.
- **Real-world considerations:** Remember stock prediction is inherently noisy. It can help to update the model regularly with new data (retrain) and to combine predictions with financial/domain knowledge.

In summary, the code reads stock data, scales it, creates input/output sequences, builds a multi-layer LSTM network, trains it to minimize prediction error, and then makes future price predictions which are visualized. Each step from data loading to model building is important to get meaningful forecasts. With more data, features, and careful tuning, the model's performance can often be improved further.

Sources: The explanation draws on machine learning best practices [1] [5], standard Keras/LSTM usage [3] [4], and stock-prediction tutorials [2] [6].

---

[1] **MinMaxScaler — scikit-learn 1.6.1 documentation**
https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

2 6 7 9 **GitHub - virajbhutada/google-stock-price-forecasting-lstm: Analyzing and predicting Google's stock prices through detailed data exploration and advanced LSTM models. This project involves data preprocessing, creating time-series sequences, constructing and training LSTM networks, and evaluating their performance to forecast future stock prices utilizing Python and Machine Learning libraries.**
https://github.com/virajbhutada/google-stock-price-forecasting-lstm

3 **LSTM layer**
https://keras.io/api/layers/recurrent_layers/lstm/

4 **Difference Between Return Sequences and Return States for LSTMs in Keras - MachineLearningMastery.com**
https://www.machinelearningmastery.com/return-sequences-and-return-states-for-lstms-in-keras/

5 **Dropout layer**
https://keras.io/api/layers/regularization_layers/dropout/

8 **Stock Price Prediction with LSTM: A Guide by Analytics Vidhya**
https://www.analyticsvidhya.com/blog/2021/12/stock-price-prediction-using-lstm/