Using Deep Learning AI to Predict the Stock Market

Forecasting Stock Prices with Neural Networks containing Multivariable Inputs from Technical Analysis

Marco Santos Follow
May 4 · 9 min read ★

Photo by Vladimir Solomyani on Unsplash

Imagine being able to know when a stock is heading up or going down in the next week and then with the remaining cash you have, you would put all of your money to invest or short that stock. After playing the stock market with the knowledge of whether or not the stock will increase or decrease in value, you might end up a millionaire!

Unfortunately, this is impossible because no one can know the future. However, we *can* make estimated guesses and informed forecasts based on the information we have in the present and the past regarding any stock. An estimated guess from past movements and patterns in stock price is called **Technical Analysis**. We can use Technical Analysis (*TA*)to predict a stock's price direction, however, this is not 100% accurate. In fact, some traders criticize TA and have said that it is just as effective in predicting the future as Astrology. But there are other traders out there who swear by it and have established long successful trading careers.

In our case, the Neural Network we will be using will utilize TA to help it make informed predictions. The specific Neural Network we will implement is called a **Recurrent Neural Network — LSTM**. Previously we utilized an RNN to predict Bitcoin prices (*see article below*):

Predicting Bitcoin Prices with Deep Learning

Using Neural Networks to Forecast Bitcoin Prices

towardsdatascience.com

In the article, we explored the usage of LSTM to predict Bitcoin prices. We delved a little bit into the background of an LSTM model and gave instructions on how to program one to predict BTC prices. However, we limited the input data to Bitcoin's own price history and did not include other variables like technical indicators such as volume or moving averages.

Multivariable Input

Since the last RNN we constructed could only take in one sequence (past closing prices) to predict the future, we wanted to see if it would be possible to add even more data to the Neural Network. Maybe these other pieces of data could enhance our price forecasts? Perhaps by adding in TA indicators to our dataset, the Neural Network might be able to make much more accurate predictions? — Which is exactly what we want to accomplish here.

In the next few sections, we will be constructing a new *Recurrent Neural Network* with the capability to take in not just one piece but multiple pieces of information in the form of *technical indicators* in order to forecast future prices in the stock market.

. . .

Price History and Technical Indicators

In order to use a Neural Network to predict the stock market, we will be utilizing prices from the ***SPDR S&P 500 (SPY)***. This will give us a general overview of the stock market and by using an RNN we might be able to figure out which direction the market is heading.

Downloading Price History

To retrieve the right data for our Neural Network, you will need to head over to Yahoo Finance and *download the prices for SPY*. We will be downloading five years worth of price history for SPY as a convenient `.csv` file.

Technical Indicators

After we have downloaded the price history for SPY, we can apply a Technical Analysis Python library to produce the Technical Indicator values. A more in depth look into the process from which we were able to retrieve the indicator values was covered here:

Technical Indicators on Bitcoin using Python

Utilizing Python to Create Technical Indicators for Bitcoin

towardsdatascience.com

...

The article above goes over the exact TA Python library we utilized in order to retrieve the indicator values for SPY.

Coding the Neural Network

Import Libraries

Let's begin coding out our Neural Network by first importing some libraries:

```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from datetime import timedelta
from sklearn.preprocessing import RobustScaler
plt.style.use("bmh")

# Technical Analysis library
import ta

# Neural Network library
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout

```

First, we imported some of the usual Python libraries (*numpy, pandas, etc*). Next, we imported the technical analysis library we previously utilized to create Technical Indicators for BTC (*covered in the article above*). Then, we imported the Neural Network library from **Tensorflow Keras**. After importing the necessary libraries, we'll load in the `SPY.csv` file we downloaded from *Yahoo Finance*.

. . .

Preprocessing the Data

```
## Datetime conversion
df['Date'] = pd.to_datetime(df.Date)

# Setting the index
df.set_index('Date', inplace=True)

# Dropping any NaNs
df.dropna(inplace=True)



## Technical Indicators

# Adding all the indicators
df = ta.add_all_ta_features(df, open="Open", high="High", low="Low", cl

# Dropping everything else besides 'Close' and the Indicators
df.drop(['Open', 'High', 'Low', 'Adj Close', 'Volume'], axis=1, inplace

# Only using the last 1000 days of data to get a more accurate represen
df = df.tail(1000)



## Scaling

# Scale fitting the close prices separately for inverse_transformations
close_scaler = RobustScaler()

close_scaler.fit(df[['Close']])
```

Datetime Conversion

After loading in the data, we'll need to perform some preprocessing in order to prepare our data for the neural network and one of the first things we'll need to do is convert the DataFrame's index into the Datetime format. Then we will set the `Date` column in our data as the index for the DF.

Creating Technical Indicators

Next, we'll create some technical indicators by using the `ta` library. To cover as much technical analysis as possible, we'll use *all* the indicators available to us from the library. Then, drop everything else besides the *indicators* and the *Closing prices* from the dataset.

Recent Data

Once we have created the technical indicator values, we can then eliminate some rows from our original dataset. We will only be including the last 1000 rows of data in order to have a more accurate representation of the current market climate.

Helper Functions

Before we start constructing the neural network, let's create some helper functions to better optimize the process. We'll explain each function in detail.

Scaling the Data

When scaling our data, there are multiple approaches to take to make sure our data is still accurately represented. It may be useful to experiment with different scalers to see their effect on model performance.

In our case, we will be utilizing `RobustScaler` to scale our data. This is done so that extreme outliers will have little effect and hopefully improve training time and overall model performance.

```python
def split_sequence(seq, n_steps_in, n_steps_out):
    """
    Splits the multivariate time sequence
    """

    # Creating a list for both variables
    X, y = [], []

    for i in range(len(seq)):

        # Finding the end of the current sequence
        end = i + n_steps_in
        out_end = end + n_steps_out

        # Breaking out of the loop if we have exceeded the dataset's l
        if out_end > len(seq):
            break

        # Splitting the sequences into: x = past prices and indicators
        seq_x, seq_y = seq[i:end, :], seq[end:out_end, 0]

        X.append(seq_x)
        y.append(seq_y)

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


def visualize_training_results(results):
    """
    Plots the loss and accuracy for the training and testing data
    """
    history = results.history
    plt.figure(figsize=(16,5))
    plt.plot(history['val_loss'])
    plt.plot(history['loss'])
    plt.legend(['val_loss', 'loss'])
    plt.title('Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.show()

    plt.figure(figsize=(16,5))
    plt.plot(history['val_accuracy'])
    plt.plot(history['accuracy'])
```

1. `split_sequence` — This function splits a multivariate time sequence. In our case, the input values are going to be the *Closing* prices and *indicators* for a stock. This will split the values into our ***X*** and ***y*** variables. The ***X*** values will contain the past closing prices and technical indicators. The ***y*** values will contain our target values (*future closing prices only*).

2. `visualize_training_results` — This function will help us evaluate the Neural Network we just created. The thing we are looking for when evaluating our NN is ***convergence***. The validation values and regular values for **Loss** and **Accuracy** must start to *align* as training progresses. If they do not converge, then that may be a sign of overfitting/underfitting. We must go back and modify the construction of the NN, which means to alter the number of layers/nodes, change the optimizer function, etc.

3. `layer_maker` — This function constructs the body of our NN. Here we can customize the number of layers and nodes. It also has a regularization option of adding Dropout layers if necessary to prevent overfitting/underfitting.

4. `validater` — This function creates a DF with predicted values for a specific range of dates. This range rolls forward with each loop. The intervals for the range are customizable. We use this DF to evaluate the model's predictions by comparing them to the actual values later on.

5. `val_rmse` — This function will return the root mean squared error (*RMSE*) of our model's predictions compared to the actual values. The value returned represents how far off our model's predictions are on average. The general goal is to reduce the

Splitting the Data

In order to appropriately format our data, we will need to split the data into two sequences. The length of these sequences can be modified but we will be using the values from the last 90 days to predict prices for the next 30 days. The `split_sequence` function will then format our data into the appropriate ***X*** and ***y*** variables where ***X*** contains the closing prices *and* indicators for the past 90 days and ***y*** contains the closing prices for the next 30 days.

RMSE of our model’s predictions.

```
# How many periods looking back to learn
n_per_in  = 90

# How many periods to predict
n_per_out = 30

# Features
n_features = df.shape[1]

# Splitting the data into appropriate sequences
X, y = split_sequence(df.to_numpy(), n_per_in, n_per_out)
```

What our NN will do with this information is *learn* how the last 90 days of closing prices and technical indicator values *affect* the next 30 days of closing prices.

. . .

Neural Network Modeling

Now we can begin constructing our Neural Network! The following code is how we construct our NN with custom layers and nodes.

```python
## Creating the NN

# Instatiating the model
model = Sequential()

# Activation
activ = "tanh"

# Input layer
model.add(LSTM(90,
               activation=activ,
               return_sequences=True,
               input_shape=(n_per_in, n_features)))

# Hidden layers
layer_maker(n_layers=1,
            n_nodes=30,
            activation=activ)

# Final Hidden layer
model.add(LSTM(60, activation=activ))

# Output layer
model.add(Dense(n_per_out))

# Model summary
model.summary()

```

This is where we begin experimenting with the parameters for:

- Number of Layers
- Number of Nodes
- Different Activation functions
- Different Optimizers

- Number of Epochs and Batch Sizes

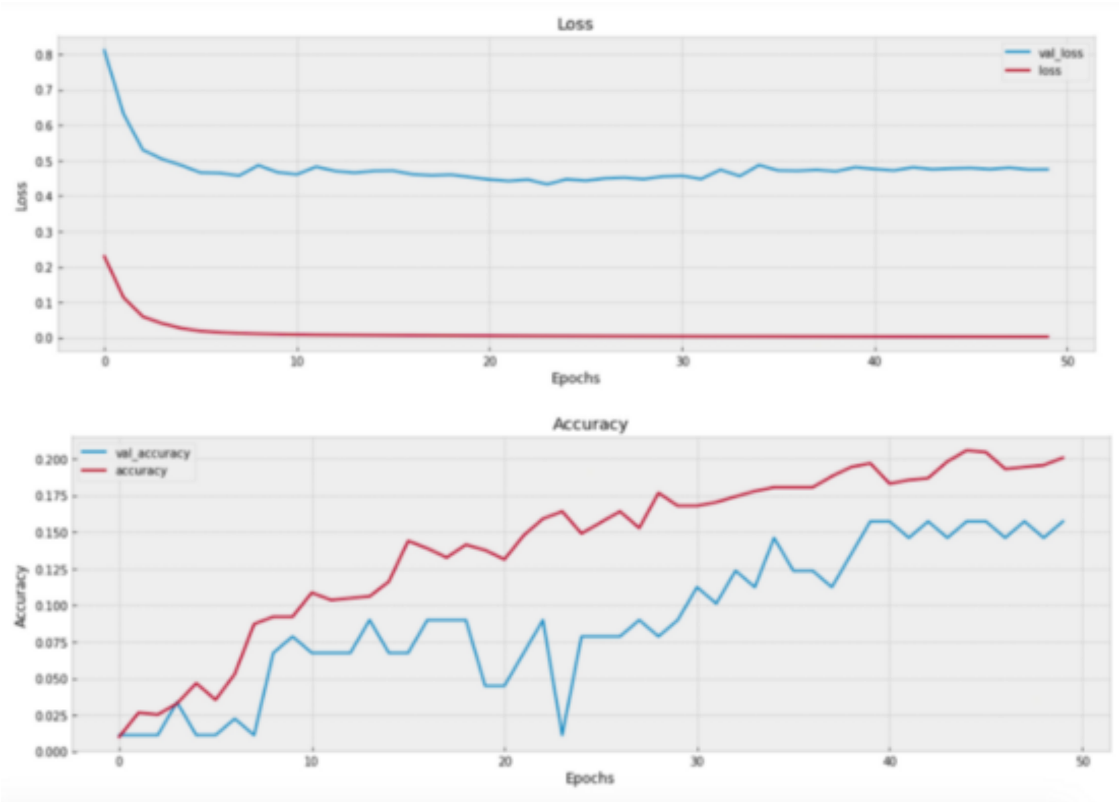
The values we input for each of these parameters will have to be explored as each value can have a significant effect on the overall model's quality. There are probably methods out there to find the optimum values for each parameter. For our case we subjectively tested out different values for each parameter and the best ones we found can be seen in the code snippet above.

If you wish to know more about the reasoning and concepts behind these variables, then it is suggested that you read our *previous article about Deep Learning and Bitcoin*.

Visualizing Loss and Accuracy

After training, we will visualize the progress of our Neural Network with our custom helper function:

```
visualize_training_results(res)
```



As our network trains, we can see that the Loss decreasing and Accuracy increasing. As a general rule, we are looking for the two lines to ***converge or align*** together as the number of epochs increases. If they do not, then that is a sign that the model is inadequate and we will need to go back and change some parameters.

Model Validation

Another way we can evaluate the quality of our model's predictions is to test it against the actual values and calculate the RMSE with our custom helper function: `val_rmse` .

```python
# Transforming the actual values to their original price
actual = pd.DataFrame(close_scaler.inverse_transform(df[["Close"]]),
                      index=df.index,
                      columns=[df.columns[0]])

# Getting a DF of the predicted values to validate against
predictions = validater(n_per_in, n_per_out)

# Printing the RMSE
print("RMSE:", val_rmse(actual, predictions))

# Plotting
plt.figure(figsize=(16,6))

# Plotting those predictions
plt.plot(predictions, label='Predicted')

# Plotting the actual values
plt.plot(actual, label='Actual')

plt.title(f"Predicted vs Actual Closing Prices")
plt.ylabel("Price")
```

Forecasting the Future

Once we are satisfied with how well our model performs, then we can use it to predict future values. This part is fairly simply relative

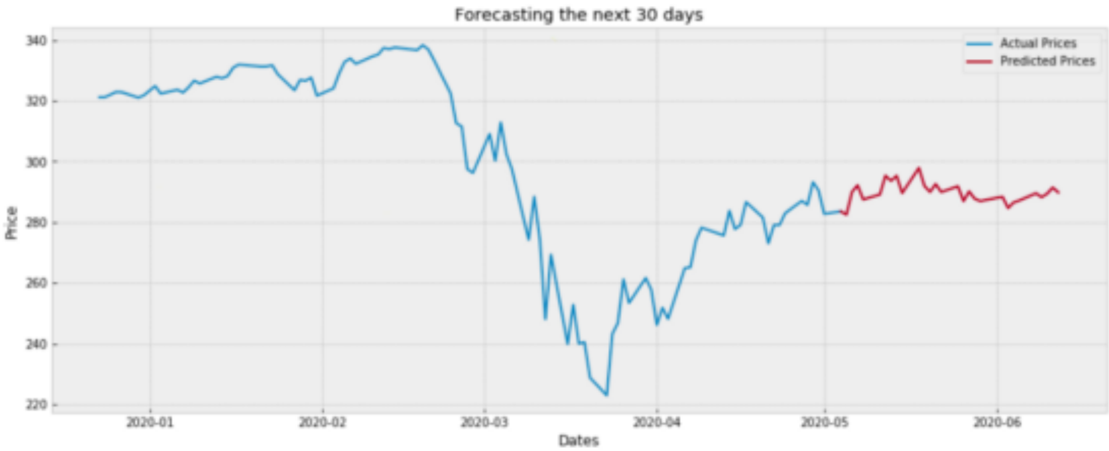
RMSE: 10.406982887153703

Here we plot the predicted values with the actual values to see how well the compare. If the plot of the predicted values are extremely off and nowhere near the actual values, then we know that our model is deficient. However, if the values appear visually close and the RMSE is ve

```
# Predicting off of the most recent days from the original DF
yhat = model.predict(np.array(df.tail(n_per_in)).reshape(1, n_per_in, 

# Transforming the predicted values back to their original format
yhat = close_scaler.inverse_transform(yhat)[0]

# Creating a DF of the predicted prices
preds = pd.DataFrame(yhat,
                     index=pd.date_range(start=df.index[-1]+timedelta(
                                         periods=len(yhat),
                                         freq="B"),
                     columns=[df.columns[0]])

# Number of periods back to plot the actual values
pers = n_per_in

# Transforming the actual values to their original price
actual = pd.DataFrame(close_scaler.inverse_transform(df[["Close"]].tail
                      index=df.Close.tail(pers).index,
                      columns=[df.columns[0]]).append(preds.head(1))

# Printing the predicted prices
print(preds)

# Plotting
plt.figure(figsize=(16,6))
plt.plot(actual, label="Actual Prices")
plt.plot(preds, label="Predicted Prices")
```

ture or moc
e last three
d
s.

Here we are just predicting off of the most recent values we have from the downloaded `.csv` file. Once we run the code we are presented with the following forecast:



SPY price forecast for the next month

just one. The quality of the model may vary from person to person depending on how much time they wish to spend on it. These predictions can be extremely useful for those wishing to gain some insight into the future price movement of a stock even though predicting the future isn't possible. But, it is likely to believe that this way is better than randomly guessing.

And there we have it! — The forecasted prices for SPY. Do what you wish with this knowledge but remember one thing: *the stock market is unpredictable*. The values predicted here are *not* certain. They may be better than just randomly guessing since the values are educated guesses based on the Technical Indicators and price patterns from the past.

*Follow me on Twitter: @_Marco_Santos_*

Resources

marcosan93/Price-Forecaster

Using Time Series models: SARIMA and FB Prophet to forecast Bitcoin prices: Using a Recurrent Neural Network: LSTM...

github.com

Predicting Bitcoin Prices with Deep Learning

Using Neural Networks to Forecast Bitcoin Prices towardsdatascience.com

Technical Indicators on Bitcoin using Python

Utilizing Python to Create Technical Indicators for Bitcoin

towardsdatascience.com

How to Develop LSTM Models for Time Series Forecasting - Machine Learning Mastery

Long Short-Term Memory networks, or LSTMs for short, can be applied to time series forecasting. There are many types of…

machinelearningmastery.com

Finance Artificial Intelligence Stock Market Technology Money