Neural Networks to Predict the Market

Vivek Palaniappan

Sep 23, 2018 · 8 min read ★

Machine Learning and deep learning have become new and effective strategies commonly used by quantitative hedge funds to maximize their profits. As an AI and finance enthusiast myself, this is exciting

news as it combines two of my areas of interest. This article will be an introduction on how to use neural networks to predict the stock market, in particular the price of a stock (or index). This post is based on **python project** in my GitHub, where you can find the full python code and how to use the program. Also, for more content like this, check out my own page: **Engineer Quant**

Neural Networks in Finance

Finance is highly nonlinear and sometimes stock price data can even seem completely random. Traditional time series methods such as ARIMA and GARCH models are effective only when the series is stationary, which is a restricting assumption that requires the series to be preprocessed by taking log returns (or other transforms). However, the main issue arises in implementing these models in a live trading system, as there is no guarantee of stationarity as new data is added.

This is combated by using neural networks, which do not require any stationarity to be used. Furthermore, neural networks by nature are effective in finding the relationships between data and using it to predict (or classify) new data.

A typical full stack data science project has the following workflow:

1. Data acquisition — this provides us the features
2. Data preprocessing — an often dreaded but necessary step to make the data usable
3. Develop and implement model — where we choose the type of

neural network and parameters

4. Backtest model — a very crucial step in any trading strategy
5. Optimization — finding suitable parameters

The input data for our neural network is the past ten days of stock price data and we use it to predict the next day's stock price data.

Data Acquisition

Fortunately, the stock price data required for this project is readily available in Yahoo Finance. The data can be acquired by either using their Python API, `pdr.get_yahoo_data(ticker, start_date, end_date` or directly from their website.

Data Preprocessing

In our case, we need to break up the data into training sets of ten prices and the next day price. I have done this by defining a class `Preprocessing`, breaking it up into train and test data and defining a method `get_train(self, seq_len)` that returns the training data (input and output) as `numpy` arrays, given a particular length of window (ten in our case). The full code is as follows:

```
def gen_train(self, seq_len):
    """
    Generates training data
    :param seq_len: length of window
    :return: X_train and Y_train
    """
    for i in
```

```
range((len(self.stock_train)//seq_len)*seq_len -
seq_len - 1):
        x = np.array(self.stock_train.iloc[i: i +
seq_len, 1])
        y = np.array([self.stock_train.iloc[i +
seq_len + 1, 1]], np.float64)
        self.input_train.append(x)
        self.output_train.append(y)
    self.X_train = np.array(self.input_train)
    self.Y_train = np.array(self.output_train)
```

Similarly, for the test data, I defined a method that returns the test data `X_test` and `Y_test`.

Neural Network Models

For this project, I have used two neural network models: the Multilayer Perceptron (MLP) and the Long Short Term Model (LSTM). I will give a short introduction into how these models work, but to read through how MLPs work, check out this **article**. For LSTMs, check out this excellent **article** by Jakob Aungiers.

MLPs are simplest form of neural networks, where an input is fed into the model, and using certain weights, the values are fed forward through the hidden layers to produce the output. The learning comes from backpropagating through the hidden layers to change the value of the weights between each neuron. An issue with MLPs is the lack of 'memory'. There is no sense of what happened in previous training data and how that might and should affect the new training data. In the context of our model, the difference between the ten days of data in one dataset and another dataset might be of importance (for

example) but MLPs do not have the ability to analyse these relationships.

This is where LSTMs, or in general Recurrent Neural Networks (RNNs) come in. RNNs have the ability of storing certain information about the data for later use and this extends the network's capability in analyzing the complex structure of the relationships between stock price data. A problem with RNNs is the vanishing gradient problem. This is due to the fact that when the number of layers increases, the learning rate (value less that one) is multiplied several times, and that causes the gradient to keep decreasing. This is combated by LSTMs, making them more effective.

Implementing Models

To implement the models, I have chosen `keras` because it uses the idea of adding layers to the network instead of defining the entire network at once. This opens us up to quick alteration of the number of layers and type of layers, which is handy when optimizing the network.

An important step in using the stock price data is to normalize the data. This would usually mean that you minus the average and divide by standard deviation but in our case, we want to be able to use this system on live trade over a period of time. So taking the statistical moments might not be the most accurate way to normalize the data. So I have merely divided the entire data by 200 (an arbitrary number that makes everything small). Although it seems as though the

normalization was plucked out of thin air, it is still effective in making sure the weights in the neural network do not grow too large.

Let us begin with the simpler MLP. In `keras` this is done by making a sequential model and adding dense layers on top of it. The full code is as follows:

```
model = tf.keras.models.Sequential()

model.add(tf.keras.layers.Dense(100,
activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(100,
activation=tf.nn.relu))
model.add(tf.keras.layers.Dense(1,
activation=tf.nn.relu))

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

This is where the elegance of keras really shows. Just with those five lines of code, we have created a MLP with two hidden layers each with a hundred neurons. A little word about the optimizer. Adam optimizer is gaining popularity in the machine learning community because it is a more efficient algorithm to optimize compared to traditional stochastic gradient descent. The advantages are best understood by looking at the advantages of two other extensions of stochastic gradient descent:

- **Adaptive Gradient Algorithm** (AdaGrad) that maintains a per-parameter learning rate that improves performance on problems

with sparse gradients (e.g. natural language and computer vision problems).

- **Root Mean Square Propagation** (RMSProp) that also maintains per-parameter learning rates that are adapted based on the average of recent magnitudes of the gradients for the weight (e.g. how quickly it is changing). This means the algorithm does well on online and non-stationary problems (e.g. noisy).

Adam can be thought of as combining the benefits of the above extensions and that is why I have chosen to use Adam as my optimizer.

Now we need to fit the model with our training data. Again, keras makes it simple with only requiring the following code:

```
model.fit(X_train, Y_train, epochs=100)
```

Once we fit our model, we need to evaluate it against our test data to see how well it performed. This is done by

```
model.evaluate(X_test, Y_test)
```

You can use the information from the evaluation to assess the ability of the model to predict the stock prices.

For the LSTM model, the procedure is similar, hence I will post the code below, leaving the explaining for you to read up on:

```
model = tf.keras.Sequential()
model.add(tf.keras.layers.LSTM(20, input_shape=(10,
1), return_sequences=True))
model.add(tf.keras.layers.LSTM(20))
model.add(tf.keras.layers.Dense(1,
activation=tf.nn.relu))

model.compile(optimizer="adam",
loss="mean_squared_error")

model.fit(X_train, Y_train, epochs=50)

model.evaluate(X_test, Y_test)
```

One important point to note is the requirement by keras for the input data to be of certain dimensions, determined by your model. It is crucial that you reshape your data using numpy.

Backtesting the Model

Now that we have fitted our models using our training data and evaluated it using our test data, we can take the assessment a step further by backtesting the model on new data. This is done simply by

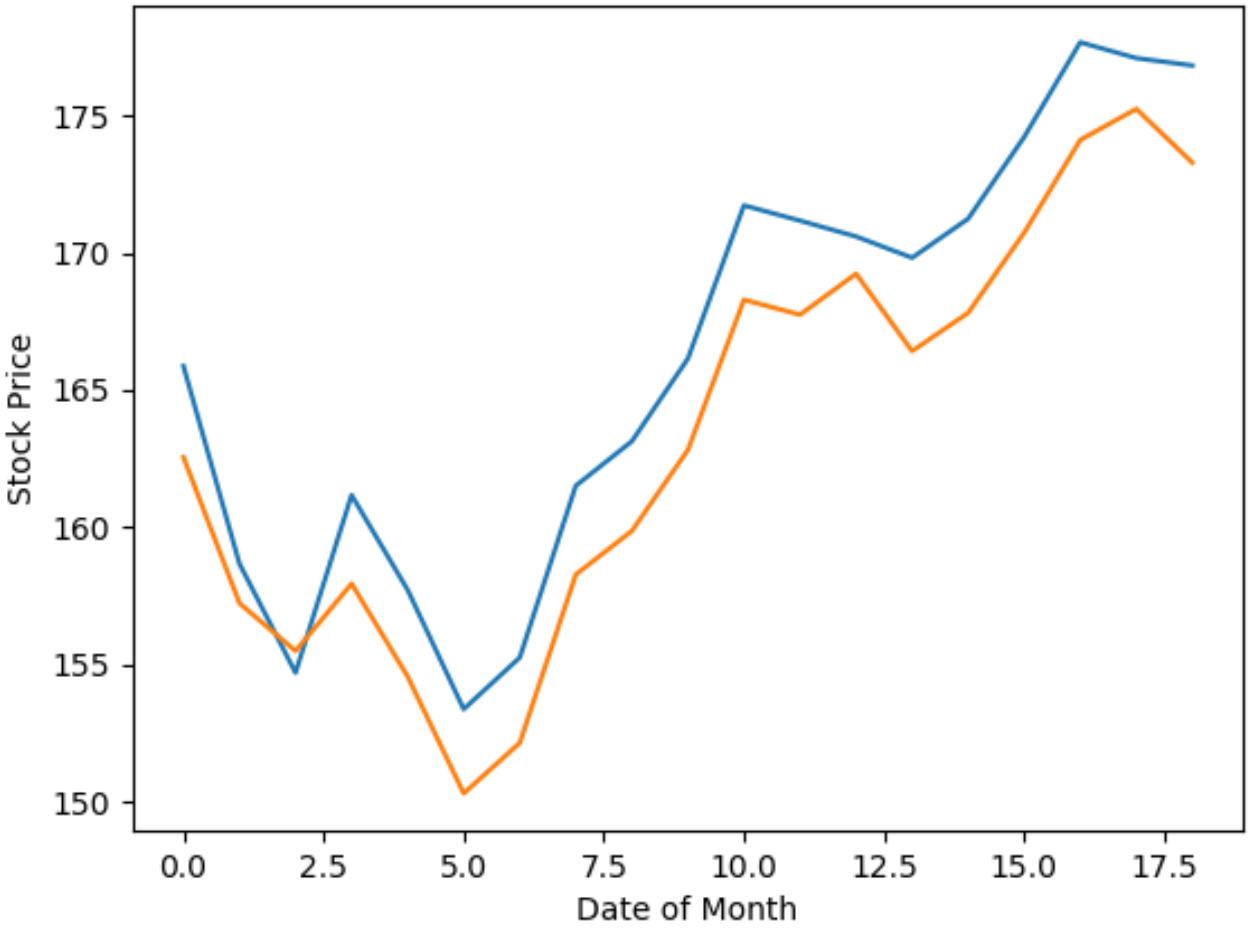
```
def back_test(strategy, seq_len, ticker, start_date,
end_date, dim):
    """
    A simple back test for a given date period
    :param strategy: the chosen strategy. Note to
```

```
    have already formed the model, and fitted with
    training data.
        :param seq_len: length of the days used for
    prediction
        :param ticker: company ticker
        :param start_date: starting date
        :type start_date: "YYYY-mm-dd"
        :param end_date: ending date
        :type end_date: "YYYY-mm-dd"
        :param dim: dimension required for strategy: 3dim
    for LSTM and 2dim for MLP
        :type dim: tuple
        :return: Percentage errors array that gives the
    errors for every test in the given date range
        """
        data = pdr.get_data_yahoo(ticker, start_date,
    end_date)
        stock_data = data["Adj Close"]
        errors = []
        for i in range((len(stock_data)//10)*10 - seq_len
    - 1):
            x = np.array(stock_data.iloc[i: i + seq_len,
    1]).reshape(dim) / 200
            y = np.array(stock_data.iloc[i + seq_len + 1,
    1]) / 200
            predict = strategy.predict(x)
            while predict == 0:
                predict = strategy.predict(x)
            error = (predict - y) / 100
            errors.append(error)
            total_error = np.array(errors)
        print(f"Average error = {total_error.mean()}")
```

However, this backtesting is a simplified version and not a full blown backtest system. For full blown backtest systems, you will need to consider factors such as survivorship bias, look ahead bias, market regime change and transaction costs. Since this is merely an educational project, a simple backtest suffices. However, if you have

questions about setting up a full backtest system, then feel free to contact me.

The following shows how my LSTM model performed when predicting the Apple stock price over the month of February



For a simple LSTM model with no optimization, that is quite good prediction. It really shows us how robust neural networks and machine learning models are in modelling complex relationships between parameters.

Hyperparameter Tuning

Optimizing the neural network model is often important to improve the performance of the model in out of sample testing. I have not included the tuning in my open source version of the project, as I want it to be a challenge to those reading it to go ahead and try to optimize the model to make it perform better. For those who do not know about optimizing, it involves finding the hyperparameters that maximize the performance of the model. There are several ways in which you can search for these ideal hyperparameters, from grid search to stochastic methods. I strongly feel that learning to optimize models can take your machine learning knowledge to new level, and hence, I am going to challenge you to come up with a optimized model that beats my performance shown in graph above.

Conclusion

Machine learning is constantly evolving with new methods being developed every day. It is crucial that we update our knowledge constantly and the best way to do so is by building models for fun projects like stock price prediction. Although the LSTM model above is not good enough to be used in live trading, the foundations built by developing such a model can help us build better models that might one day be used in our trading system.

Machine Learning | Neural Networks | Stock Market | Predictions | Quantitative Finance

Medium

About Help Legal

Read more on Medium.
Create a free account.