Using LSTMs For Stock Market Predictions (Tensorflow)

Thushan Ganegedara · Follow
Published in Towards Data Science
13 min read · May 18, 2018

In this tutorial, you will see how you can use a time-series model known as Long Short-Term Memory. LSTM models are powerful, especially for retaining a long-term memory, by design, as you will see later. You'll tackle the following topics in this tutorial:

- Understand why would you need to be able to predict stock price movements;

- Download the data — You will be using stock market data gathered from Alphavantage/Kaggle;
- Split train-test data and also perform some data normalization;
- Motivate and briefly discuss an LSTM model as it allows to predict more than one-step ahead;
- Predict and visualize future stock market with current data

Note: But before we start, *I'm not advocating LSTMs as a highly reliable model that exploits the patterns in stock data perfectly, or can be used blindly without any human-in-the-loop*. I did this as an experiment, in a pure machine learning interest. In my opinion, the model has observed certain patterns in the data, thus giving it the ability to correctly predict the stock movements most of the time. But it is up to you to decide if this model can be used for practical purposes or not.

Why Do You Need Time Series Models?

You would like to model stock prices correctly, so as a stock buyer you can reasonably decide when to buy stocks and when to sell them to make a profit. This is where time series modeling comes in. You need good machine learning models that can look at the history of a sequence of data and correctly predict what the future elements of the sequence are going to be.

Warning: Stock market prices are highly unpredictable and volatile. This means that there are no consistent patterns in the data that allow you to model stock prices over time near-perfectly. Don't take it from me, take it from Princeton University economist Burton Malkiel, who argues in his 1973 book, "A Random Walk Down Wall Street," that *if the market is truly efficient and a share price reflects all factors immediately as soon as they're made public, a blindfolded monkey throwing darts at a newspaper stock listing should do as well as any investment professional.*

However, let's not go all the way believing that this is just a stochastic or random process and that there is no hope for machine learning. Let's see if you can at least model the data, so that the predictions you make correlate with the actual behavior of the data. In other words, you don't need the exact stock values of the future, but the stock price movements (that is, if it is going to rise of fall in the near future).

```
# Make sure that you have all these libaries available to run the code successfully
from pandas_datareader import data
import matplotlib.pyplot as plt
import pandas as pd
import datetime as dt
import urllib.request, json
import os
import numpy as np
import tensorflow as tf # This code has been tested with TensorFlow 1.6
from sklearn.preprocessing import MinMaxScaler
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Downloading the Data

You will be using data from the following sources:

1. Alpha Vantage. Before you start, however, you will first need an API key, which you can obtain for free here. After that, you can assign that key to the `api_key` variable.
2. Use the data from this page. You will need to copy the *Stocks* folder in the zip file to your project home folder.

Stock prices come in several different flavours. They are,

- Open: Opening stock price of the day
- Close: Closing stock price of the day
- High: Highest stock price of the data
- Low: Lowest stock price of the day

Getting Data from Alphavantage

You will first load in the data from Alpha Vantage. Since you're going to make use of the American Airlines Stock market prices to make your predictions, you set the ticker to `"AAL"`. Additionally, you also define a `url_string`, which will return a JSON file with all the stock market data for American Airlines within the last 20 years, and a `file_to_save`, which will be the file to which you save the data. You'll use the `ticker` variable that you defined beforehand to help name this file.

Next, you're going to specify a condition: if you haven't already saved data, you will go ahead and grab the data from the URL that you set in `url_string`; You'll store the

date, low, high, volume, close, open values to a pandas DataFrame `df` and you'll save it to `file_to_save`. However, if the data is already there, you'll just load it from the CSV.

Getting Data from Kaggle

Data found on Kaggle is a collection of csv files and you don't have to do any preprocessing, so you can directly load the data into a Pandas DataFrame. Make sure you download the data into the project home directory. So that the *Stocks* folder should be in the project home directory.

Data Exploration

Here you will print the data you collected into the DataFrame. You should also make sure that the data is sorted by date, because the order of the data is crucial in time series modeling.

```
# Sort DataFrame by date
df = df.sort_values('Date')

# Double check the result
df.head()
```

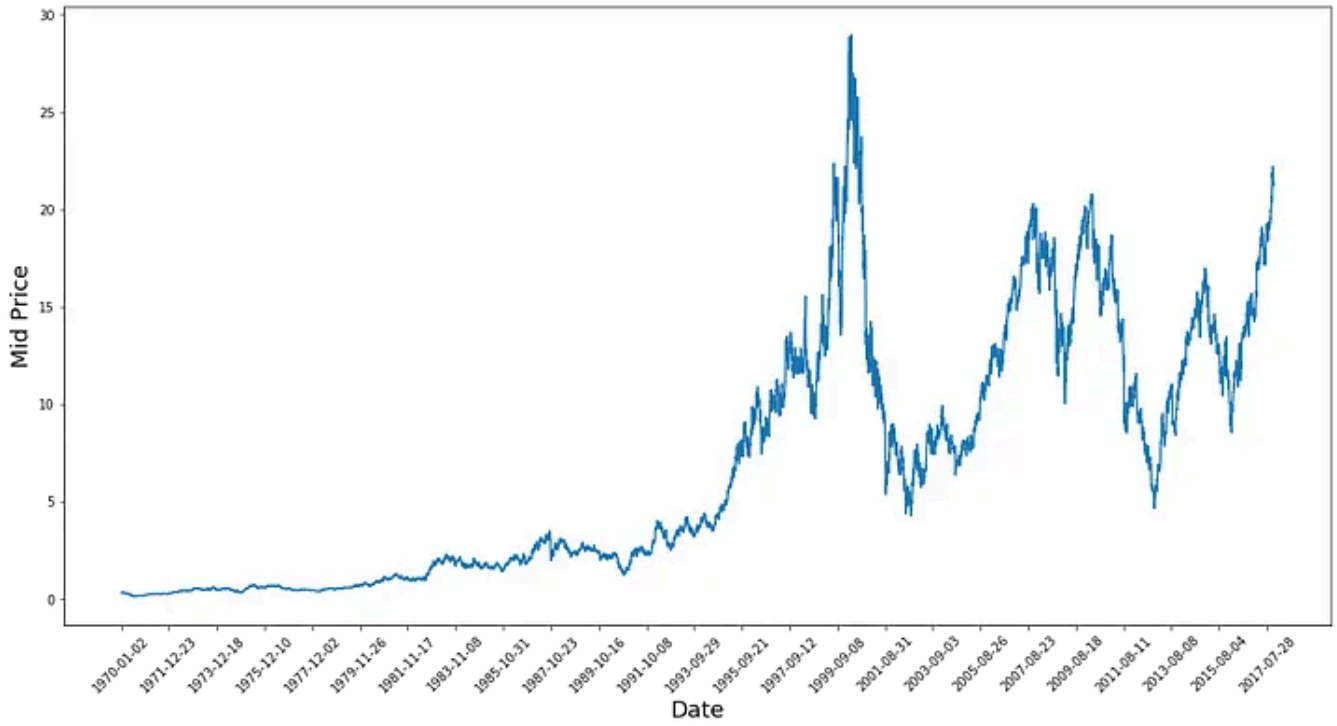
lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Data Visualization

Now let's see what sort of data you have. You want data with various patterns occurring over time.

```
plt.figure(figsize = (18,9))
plt.plot(range(df.shape[0]),(df['Low']+df['High'])/2.0)
plt.xticks(range(0,df.shape[0],500),df['Date'].loc[::500],rotation=45)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Mid Price',fontsize=18)
plt.show()
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw



This graph already says a lot of things. The specific reason I picked this company over others is that this graph is bursting with different behaviors of stock prices over time. This will make the learning more robust as well as give you a change to test how good the predictions are for a variety of situations.

Another thing to notice is that the values close to 2017 are much higher and fluctuate more than the values close to the 1970s. Therefore you need to make sure that the data behaves in similar value ranges throughout the time frame. You will take care of this during the *data normalization* phase.

Splitting Data into a Training set and a Test set

You will use the mid price calculated by taking the average of the highest and lowest recorded prices on a day.

```
# First calculate the mid prices from the highest and lowest
high_prices = df.loc[:,'High'].as_matrix()
low_prices = df.loc[:,'Low'].as_matrix()
mid_prices = (high_prices+low_prices)/2.0
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Now you can split the training data and test data. The training data will be the first 11,000 data points of the time series and rest will be test data.

```
train_data = mid_prices[:11000]
test_data = mid_prices[11000:]
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Now you need to define a scaler to normalize the data. `MinMaxScalar` scales all the data to be in the region of 0 and 1. You can also reshape the training and test data to be in the shape `[data_size, num_features]`.

```
# Scale the data to be between 0 and 1
# When scaling remember! You normalize both test and train data with respect to training
# Because you are not supposed to have access to test data
scaler = MinMaxScaler()
train_data = train_data.reshape(-1,1)
test_data = test_data.reshape(-1,1)
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Due to the observation you made earlier, that is, different time periods of data have different value ranges, you normalize the data by splitting the full series into windows. If you don't do this, the earlier data will be close to 0 and will not add much value to the learning process. Here you choose a window size of 2500.

Tip: when choosing the window size make sure it's not too small, because when you perform windowed-normalization, it can introduce a break at the very end of each window, as each window is normalized independently.

In this example, 4 data points will be affected by this. But given you have 11,000 data points, 4 points will not cause any issue

```
# Train the Scaler with training data and smooth data
smoothing_window_size = 2500
for di in range(0,10000,smoothing_window_size):
    scaler.fit(train_data[di:di+smoothing_window_size,:])
    train_data[di:di+smoothing_window_size,:] = scaler.transform(train_data[di:di+smooth

# You normalize the last bit of remaining data
scaler.fit(train_data[di+smoothing_window_size:,:])
train_data[di+smoothing_window_size:,:] = scaler.transform(train_data[di+smoothing_windo
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Reshape the data back to the shape of `[data_size]`

```
# Reshape both train and test data
train_data = train_data.reshape(-1)

# Normalize test data
test_data = scaler.transform(test_data).reshape(-1)
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

You can now smooth the data using the exponential moving average. This helps you to get rid of the inherent raggedness of the data in stock prices and produce a smoother curve.

Note: We only train the MinMaxScaler with training data only, it'd be wrong to normalize test data by fitting the MinMaxScaler to test data

Note that you should only smooth training data.

```
# Now perform exponential moving average smoothing
# So the data will have a smoother curve than the original ragged data
EMA = 0.0
gamma = 0.1
for ti in range(11000):
  EMA = gamma*train_data[ti] + (1-gamma)*EMA
  train_data[ti] = EMA

# Used for visualization and test purposes
all_mid_data = np.concatenate([train_data,test_data],axis=0)
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Evauating Results

We will be using the mean squared error to compute how good our model is. The Mean Squared Error (MSE) can be calculated by taking the Squared Error between the true value at one step ahead and the predicted value and averaging it over all the predictions.

Averaging as a Stock Price Modeling Technique

In the original tutorial, I talk about how deceiving and bad averaging is for this type of a problem. The conclusion is,

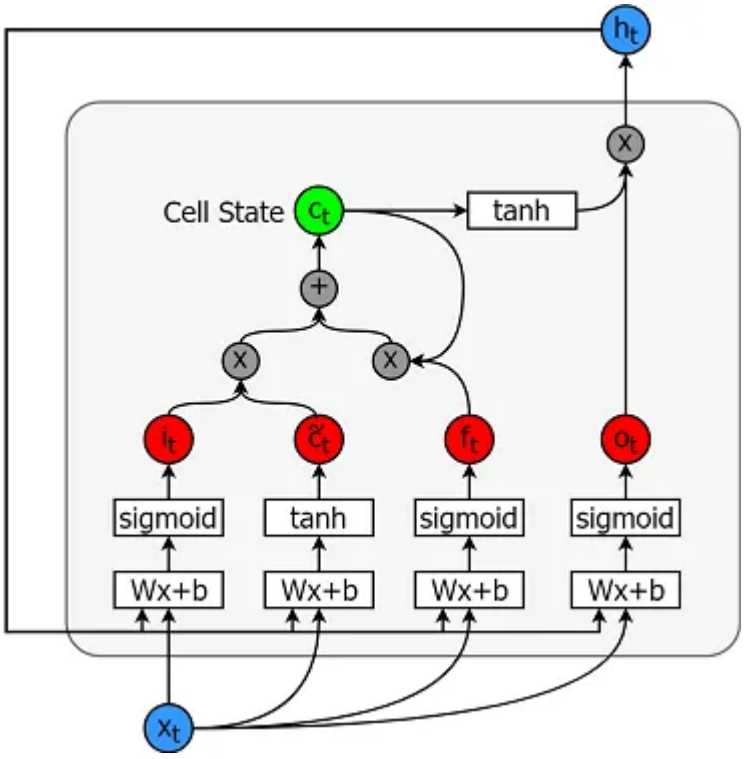
> Averaging is good to predict one time step ahead (which is not very useful for stock market predictions), but not many steps to the future. You can find more details in the original tutorial.

Introduction to LSTMs: Making Stock Movement Predictions Far into the Future

Long Short-Term Memory models are extremely powerful time-series models. They can predict an arbitrary number of steps into the future. An LSTM module (or cell) has 5 essential components which allows it to model both long-term and short-term data.

- Cell state (c_t) — This represents the internal memory of the cell which stores both short term memory and long-term memories
- Hidden state (h_t) — This is output state information calculated w.r.t. current input, previous hidden state and current cell input which you eventually use to predict the future stock market prices. Additionally, the hidden state can decide to only retrieve the short or long-term or both types of memory stored in the cell state to make the next prediction.
- Input gate (i_t) — Decides how much information from current input flows to the cell state
- Forget gate (f_t) — Decides how much information from the current input and the previous cell state flows into the current cell state
- Output gate (o_t) — Decides how much information from the current cell state flows into the hidden state, so that if needed LSTM can only pick the long-term memories or short-term memories and long-term memories

A cell is pictured below.



And the equations for calculating each of these entities are as follows.

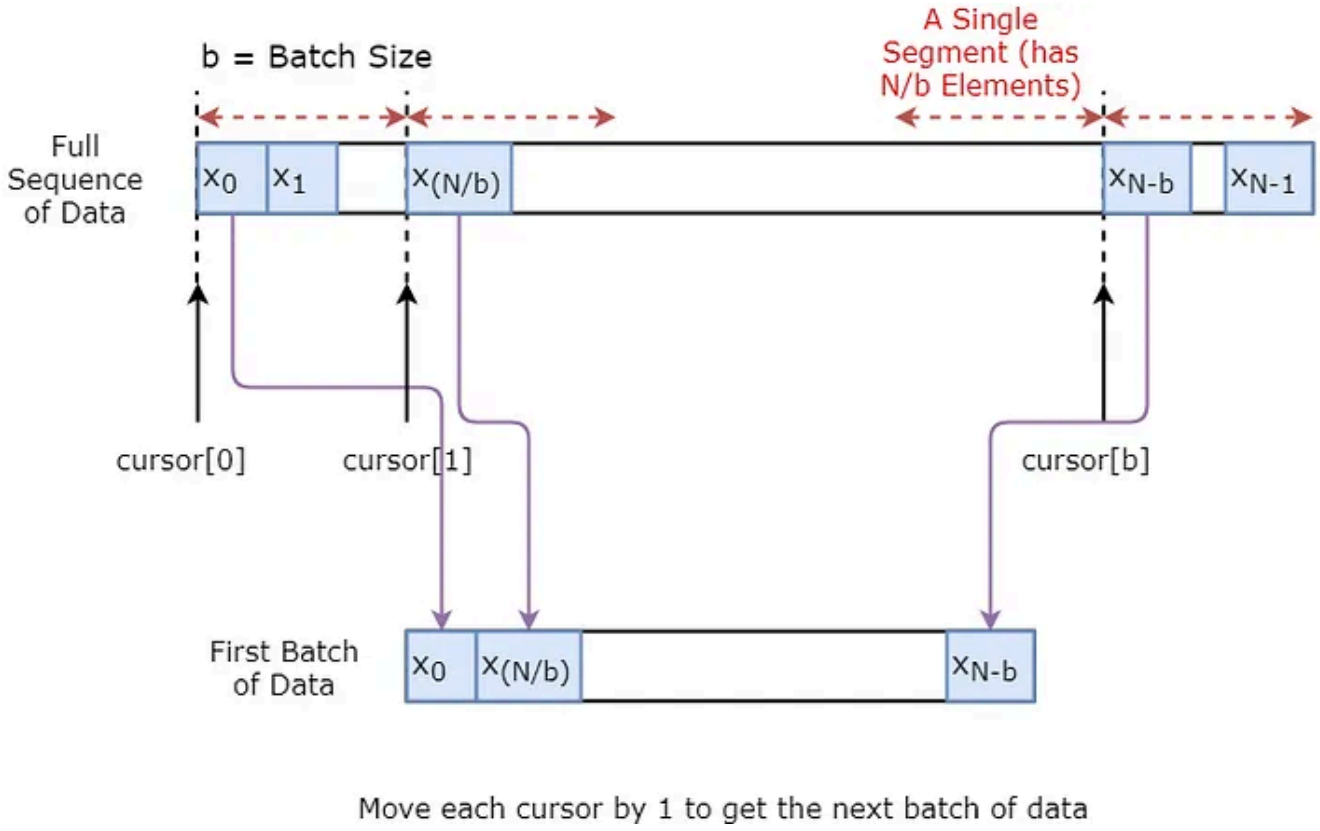
- i_*t* = σ(*W*{ix} * x_*t* + *W*{ih} * h_{t-1}+b_i)
- \tilde{c}_*t* = σ(*W*{cx} * x_*t* + *W*{ch} * h_{t-1} + b_c)
- f_*t* = σ(*W*{fx} * x*t* + *W*{fh} * h_{t-1}+b_f)
- c_t = f_*t* * *c*{t-1} + i_t * \tilde{c}_t
- o_*t* = σ(*W*{ox} * x*t* + *W*{oh} * h_{t-1}+b_o)
- h_t = o_t * tanh(c_t)

For a better (more technical) understanding about LSTMs you can refer to this article.

Data Generator

Below you illustrate how a batch of data is created visually. The basic idea is that we divide the data sequence to N/b segments, so that each segment is size b. Then we define cursors, 1 for each segment. Then to sample a single batch of data, we get one input (current segment cursor index) and one true prediction (randomly sampled one between [current segment cursor + 1, current segment cursor + 5]). Note that we are not always getting the value next to the input, as its prediction. This is a step taken to reduce overfitting. At the end of each sampling, we increase the

cursor by 1. You can find more information about the data generation in the original tutorial.



Defining Hyperparameters

In this section, you'll define several hyperparameters. `D` is the dimensionality of the input. It's straightforward, as you take the previous stock price as the input and predict the next one, which should be `1`.

Then you have `num_unrollings`, denotes how many continuous time steps you consider for a single optimization step. The larger the better.

Then you have the `batch_size`. Batch size is how many data samples you consider in a single time step. The larger the better, because more visibility of data you have at a given time.

Next you define `num_nodes` which represents the number of hidden neurons in each cell. You can see that there are three layers of LSTMs in this example.

```
D = 1 # Dimensionality of the data. Since your data is 1-D this would be 1
num_unrollings = 50 # Number of time steps you look into the future.
batch_size = 500 # Number of samples in a batch
num_nodes = [200,200,150] # Number of hidden nodes in each layer of the deep LSTM stack
n_layers = len(num_nodes) # number of layers
dropout = 0.2 # dropout amount

tf.reset_default_graph() # This is important in case you run this multiple times
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Defining Inputs and Outputs

Next you define placeholders for training inputs and labels. This is very straightforward as you have a list of input placeholders, where each placeholder contains a single batch of data. And the list has `num_unrollings` placeholders, that will be used at once for a single optimization step.

```
# Input data.
train_inputs, train_outputs = [],[]

# You unroll the input over time defining placeholders for each time step
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.float32, shape=[batch_size,D],name='train_inpu
    train_outputs.append(tf.placeholder(tf.float32, shape=[batch_size,1], name = 'train_
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Defining Parameters of the LSTM and Regression layer

You will have a three layers of LSTMs and a linear regression layer, denoted by `w` and `b`, that takes the output of the last Long Short-Term Memory cell and output the prediction for the next time step. You can use the `MultiRNNCell` in TensorFlow to encapsulate the three `LSTMCell` objects you created. Additionally, you can have the dropout implemented LSTM cells, as they improve performance and reduce overfitting.

```
lstm_cells = [
    tf.contrib.rnn.LSTMCell(num_units=num_nodes[li],
                            state_is_tuple=True,
                            initializer= tf.contrib.layers.xavier_initializer()
                           )
 for li in range(n_layers)]

drop_lstm_cells = [tf.contrib.rnn.DropoutWrapper(
    lstm, input_keep_prob=1.0,output_keep_prob=1.0-dropout, state_keep_prob=1.0-dropout
) for lstm in lstm_cells]
drop_multi_cell = tf.contrib.rnn.MultiRNNCell(drop_lstm_cells)
multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)

w = tf.get_variable('w',shape=[num_nodes[-1], 1], initializer=tf.contrib.layers.xavier_
b = tf.get_variable('b',initializer=tf.random_uniform([1],-0.1,0.1))
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Calculating LSTM output and Feeding it to the regression layer to get final prediction

In this section, you first create TensorFlow variables (`c` and `h`) that will hold the cell state and the hidden state of the Long Short-Term Memory cell. Then you transform the list of `train_inputs` to have a shape of `[num_unrollings, batch_size, D]` , this is needed for calculating the outputs with the `tf.nn.dynamic_rnn` function. You then calculate the LSTM outputs with the `tf.nn.dynamic_rnn` function and split the output back to a list of `num_unrolling` tensors. the loss between the predictions and true stock prices.

```python
# Create cell state and hidden state variables to maintain the state of the LSTM
c, h = [],[]
initial_state = []
for li in range(n_layers):
  c.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  h.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  initial_state.append(tf.contrib.rnn.LSTMStateTuple(c[li], h[li]))

# Do several tensor transofmations, because the function dynamic_rnn requires the outpu
# a specific format. Read more at: https://www.tensorflow.org/api_docs/python/tf/nn/dyn
all_inputs = tf.concat([tf.expand_dims(t,0) for t in train_inputs],axis=0)

# all_outputs is [seq_length, batch_size, num_nodes]
all_lstm_outputs, state = tf.nn.dynamic_rnn(
    drop_multi_cell, all_inputs, initial_state=tuple(initial_state),
    time_major = True, dtype=tf.float32)

all_lstm_outputs = tf.reshape(all_lstm_outputs, [batch_size*num_unrollings,num_nodes[-1

all_outputs = tf.nn.xw_plus_b(all_lstm_outputs,w,b)

split_outputs = tf.split(all_outputs,num_unrollings,axis=0)
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Loss Calculation and Optimizer

Now, you'll calculate the loss. However, you should note that there is a unique characteristic when calculating the loss. For each batch of predictions and true outputs, you calculate the Mean Squared Error. And you sum (not average) all these mean squared losses together. Finally, you define the optimizer you're going to use to optimize the neural network. In this case, you can use Adam, which is a very recent and well-performing optimizer.

```
 When calculating the loss you need to be careful about the exact form, because you cal
# loss of all the unrolled steps at the same time
# Therefore, take the mean error or each batch and get the sum of that over all the unr

print('Defining training Loss')
loss = 0.0
with tf.control_dependencies([tf.assign(c[li], state[li][0]) for li in range(n_layers)]
                              [tf.assign(h[li], state[li][1]) for li in range(n_layers)]
  for ui in range(num_unrollings):
    loss += tf.reduce_mean(0.5*(split_outputs[ui]-train_outputs[ui])**2)

print('Learning rate decay operations')
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)
tf_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)
tf_min_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)

learning_rate = tf.maximum(
    tf.train.exponential_decay(tf_learning_rate, global_step, decay_steps=1, decay_rate
    tf_min_learning_rate)

# Optimizer.
print('TF Optimization operations')
optimizer = tf.train.AdamOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(
    zip(gradients, v))

print('\tAll done')
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Here you define the prediction related TensorFlow operations. First, define a placeholder for feeding in the input (`sample_inputs`), then similar to the training stage, you define state variables for prediction (`sample_c` and `sample_h`). Finally you calculate the prediction with the `tf.nn.dynamic_rnn` function and then sending the output through the regression layer (`w` and `b`). You also should define the `reset_sample_state` operation, which resets the cell state and the hidden state. You should execute this operation at the start, every time you make a sequence of predictions.

```
print('Defining prediction related TF functions')

sample_inputs = tf.placeholder(tf.float32, shape=[1,D])

# Maintaining LSTM state for prediction stage
sample_c, sample_h, initial_sample_state = [],[],[]
for li in range(n_layers):
  sample_c.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  sample_h.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  initial_sample_state.append(tf.contrib.rnn.LSTMStateTuple(sample_c[li],sample_h[li]))

reset_sample_states = tf.group(*[tf.assign(sample_c[li],tf.zeros([1, num_nodes[li]])) f
                               *[tf.assign(sample_h[li],tf.zeros([1, num_nodes[li]])) f

sample_outputs, sample_state = tf.nn.dynamic_rnn(multi_cell, tf.expand_dims(sample_inpu
                                   initial_state=tuple(initial_sample_state),
                                   time_major = True,
                                   dtype=tf.float32)

with tf.control_dependencies([tf.assign(sample_c[li],sample_state[li][0]) for li in ran
                              [tf.assign(sample_h[li],sample_state[li][1]) for li in ra
  sample_prediction = tf.nn.xw_plus_b(tf.reshape(sample_outputs,[1,-1]), w, b)

print('\tAll done')
```

lstm_stock_market_prediction.py hosted with ❤ by GitHub view raw

Running the LSTM

Here you will train and predict stock price movements for several epochs and see whether the predictions get better or worse over time. You follow the following procedure. I'm not sharing the code as I'm sharing the link to the full Jupyter notebook.

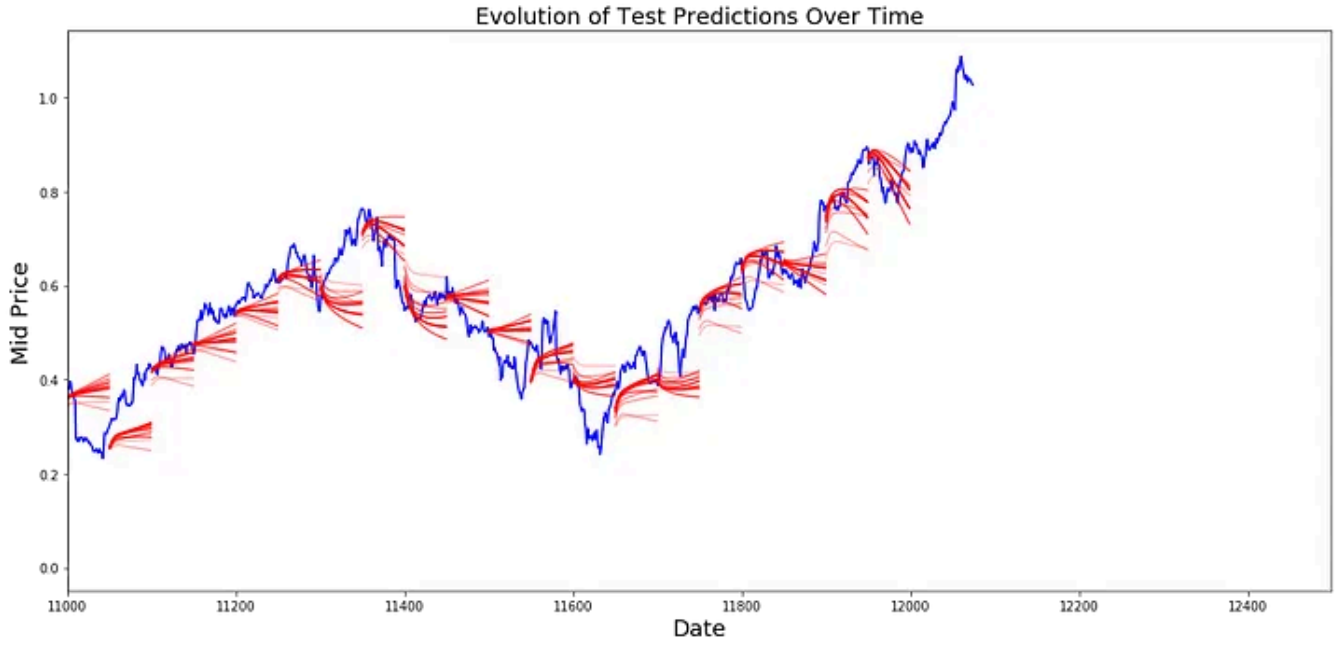
Define a test set of starting points (`test_points_seq`) on the time series to evaluate the model on

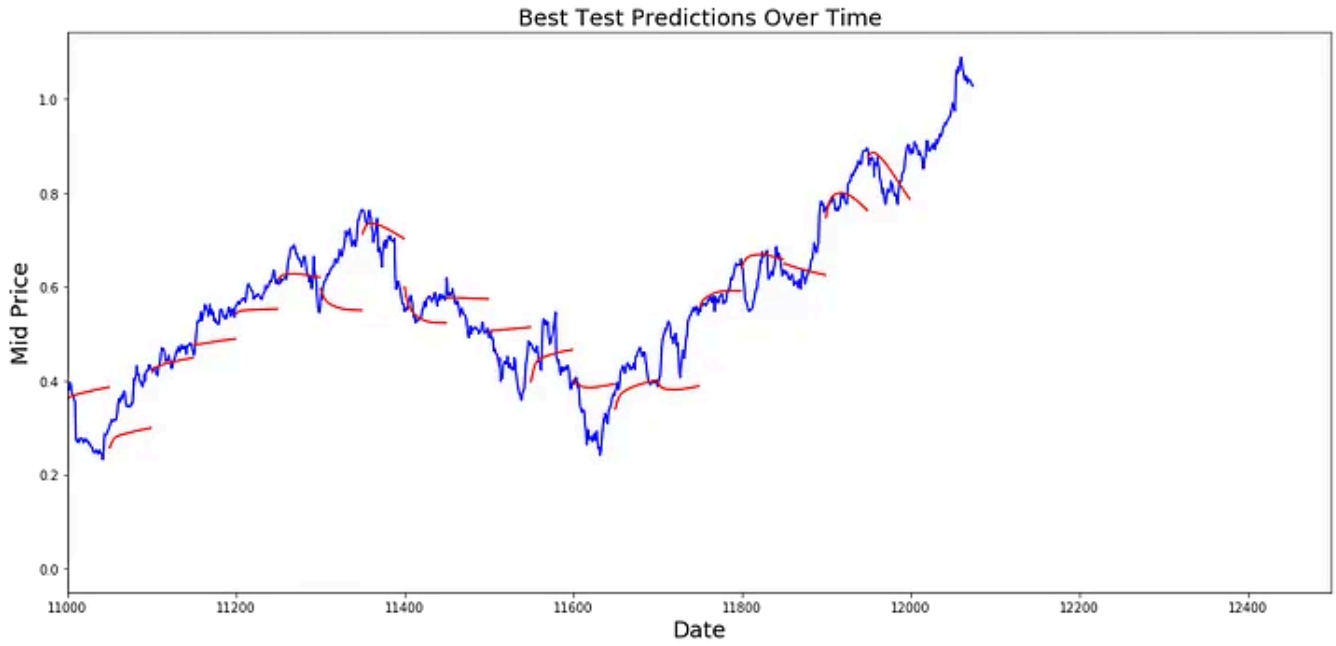
> *For each epoch
>
> **For full sequence length of training data
>
> ***Unroll a set of `num_unrollings` batches
>
> ***Train the neural network with the unrolled batches

> **Calculate the average training loss
>
> **For each starting point in the test set
>
> ***Update the LSTM state by iterating through the previous `num_unrollings` data points found before the test point
>
> ***Make predictions for `n_predict_once` steps continuously, using the previous prediction as the current input
>
> ***Calculate the MSE loss between the `n_predict_once` points predicted and the true stock prices at those time stamps

Visualizing the Predictions

You can see how the MSE loss is going down with the amount of training. This is good sign that the model is learning something useful. To quantify your findings, you can compare the network's MSE loss to the MSE loss you obtained when doing the standard averaging (0.004). You can see that the LSTM is doing better than the standard averaging. And you know that standard averaging (though not perfect) followed the true stock prices movements reasonably.





Though not perfect, LSTMs seem to be able to predict stock price behavior correctly most of the time. Note that you are making predictions roughly in the range of 0 and 1.0 (that is, not the true stock prices). This is okay, because you're predicting the stock price movement, not the prices themselves.

Conclusion

I'm hoping that you found this tutorial useful. I should mention that this was a rewarding experience for me. In this tutorial, I learnt how difficult it can be to device a model that is able to correctly predict stock price movements. You started with a motivation for why you need to model stock prices. This was followed by an explanation and code for downloading data. Then you looked at two averaging techniques that allow you to make predictions one step into the future. You next saw that these methods are futile when you need to predict more than one step into the

future. Thereafter you discussed how you can use LSTMs to make predictions many steps into the future. Finally you visualized the results and saw that your model (though not perfect) is quite good at correctly predicting stock price movements.

Here, I'm stating several takeaways of this tutorial.

1. Stock price/movement prediction is an extremely difficult task. Personally I don't think *any of the stock prediction models out there shouldn't be taken for granted and blindly rely on them*. However models might be able to predict stock price movement correctly most of the time, but not always.

2. Do not be fooled by articles out there that shows predictions curves that perfectly overlaps the true stock prices. This can be replicated with a simple averaging technique and in practice it's useless. A more sensible thing to do is predicting the stock price movements.

3. The model's hyperparameters are extremely sensitive to the results you obtain. So a very good thing to do would be to run some hyperparameter optimization technique (for example, Grid search / Random search) on the hyperparameters. Here I list some of the most critical hyperparameters; *the learning rate of the optimizer, number of layers and the number of hidden units in each layer, the optimizer (I found Adam to perform the best), type of the model (GRU/LSTM/LSTM with peepholes)*

4. In this tutorial you did something faulty (due to the small size of data)! That is you used the test loss to decay the learning rate. This indirectly leaks information about test set into the training procedure. A better way of handling this is to have a separate validation set (apart from the test set) and decay learning rate with respect to performance of the validation set.

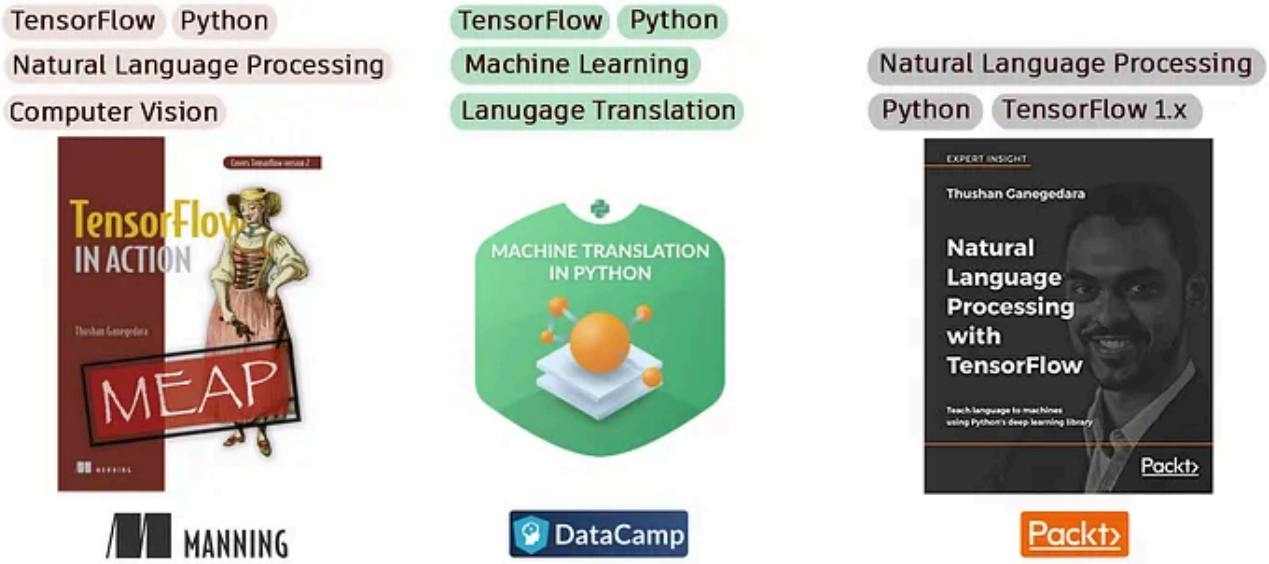
Jupyter Notebook: Here

References

I referred to this repository to get an understanding about how to use LSTMs for stock predictions. But details can be vastly different from the implementation found in the reference.

Want to get better at deep networks and TensorFlow?

Checkout my work on the subject.



[1] (Book) TensorFlow 2 in Action — Manning

[2] (Video Course) Machine Translation in Python — DataCamp

[3] (Book) Natural Language processing in TensorFlow 1 — Packt

Machine Learning | Deep Learning | TensorFlow | Lstm | Finance

Follow

Written by Thushan Ganegedara

2.9K Followers · Writer for Towards Data Science

Google Dev Expert | ML @ Canva | Author📗| PhD👨🏾‍🎓. Youtube: bit.ly/deeplearninghero | LinkedIN: thushan.ganegedara | Medium: bit.ly/support-thushan

More from Thushan Ganegedara and Towards Data Science

Thushan Ganegedara in Towards Data Science

Intuitive Guide to Understanding KL Divergence

I'm starting a new series of blog articles following a beginner friendly approach to understanding some of the challenging concepts in...

11 min read · May 1, 2018

2.2K 14

Tim Sumner in Towards Data Science

A New Coefficient of Correlation

What if you were told there exists a new way to measure the relationship between two variables just like correlation except possibly...

10 min read · Mar 31, 2024

3.4K 43

Youness Mansar in Towards Data Science

Meet the NiceGUI: Your Soon-to-be Favorite Python UI Library

Build custom web apps easily and quickly

8 min read · Apr 17, 2024

1.2K 5

Thushan Ganegedara in Towards Data Science

Troubleshooting GCP + CUDA/NVIDIA + Docker and Keeping it Running!

I had a Google Cloud Platform (GCP) instance which was all setup well and running fine a day ago, which was set up following my previous...

7 min read · Jan 15, 2018

119 1

See all from Thushan Ganegedara

See all from Towards Data Science

Recommended from Medium

SR

Predicting Stock Prices using LSTM and Yahoo Finance Data

Predicting stock prices is a challenging yet intriguing task in the field of machine learning. In this blog post, we'll explore how to use...

3 min read · Dec 16, 2023

57

Sciforce in Sciforce

Building a trade prediction model for a trader bot

The World Trade Organization (WTO) report points out the role of digital transformation in changing the way the world trades. Digital tools...

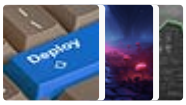
6 min read · Nov 10, 2023

444

Lists

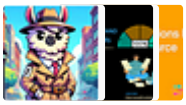
Practical Guides to Machine Learning

10 stories · 1396 saves



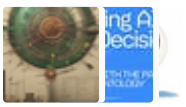
Predictive Modeling w/ Python

20 stories · 1155 saves



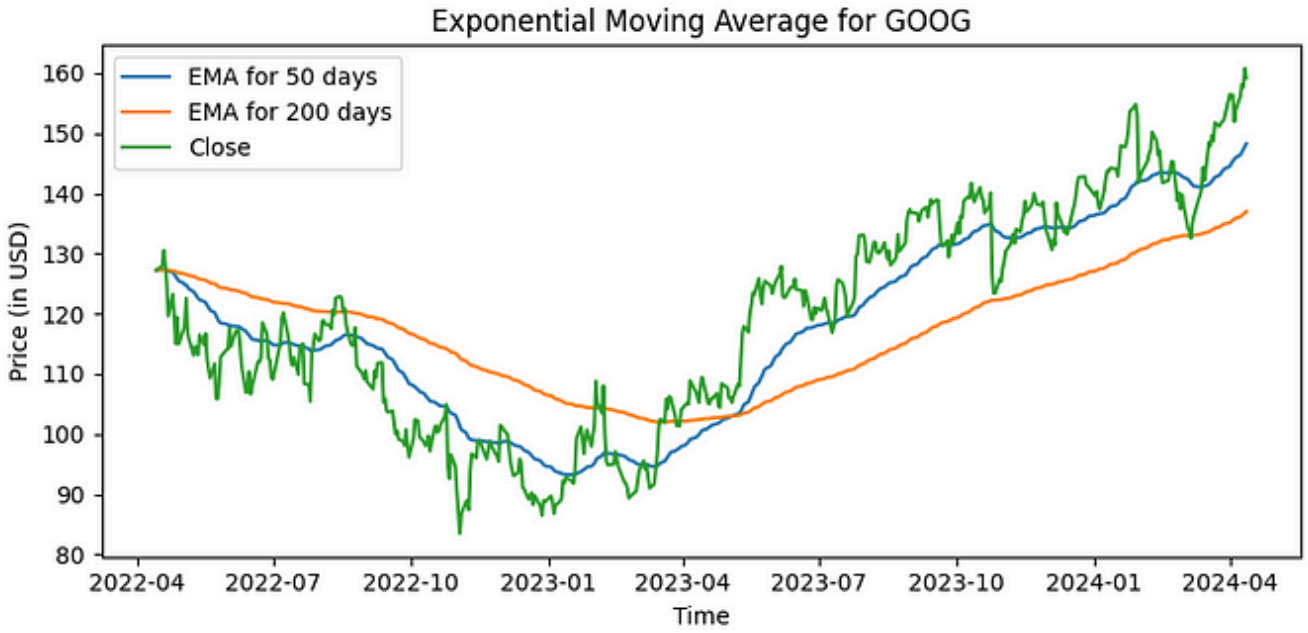
Natural Language Processing

1431 stories · 926 saves



data science and AI

40 stories · 146 saves



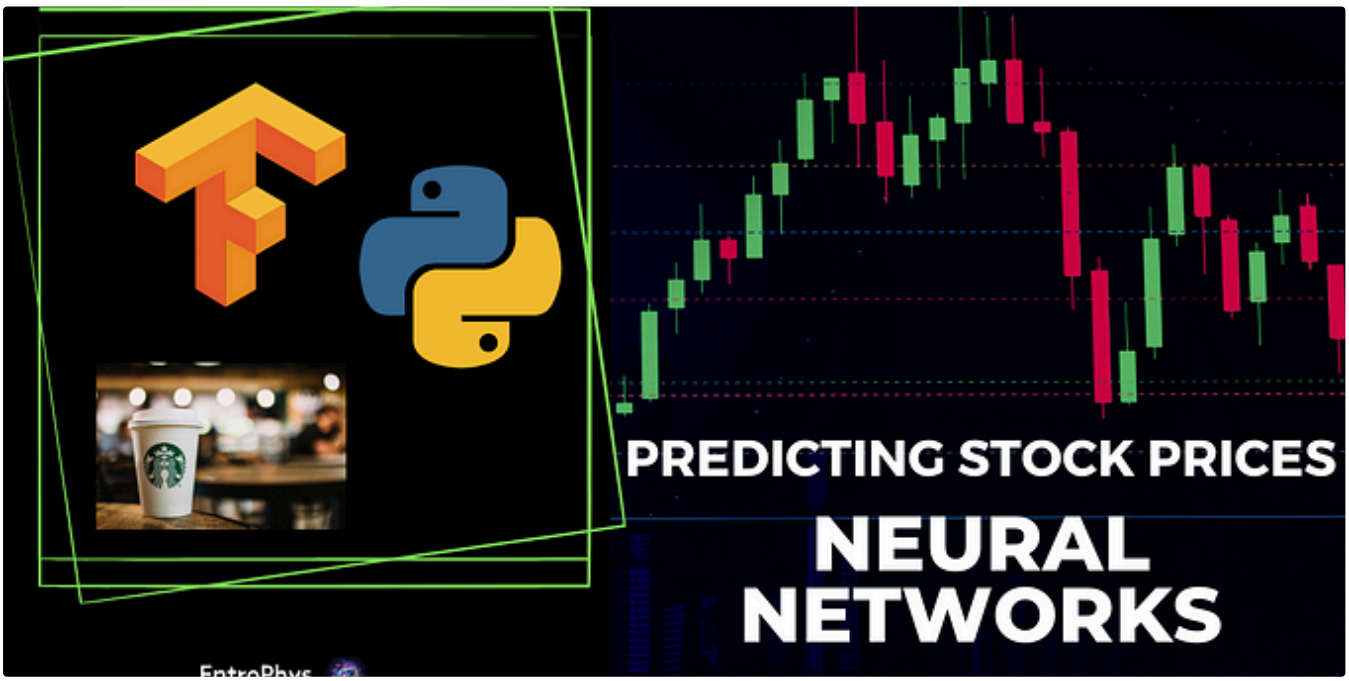
Ajk

Creating a Stock Prediction App in Python

Developed by Ayush Kulkarni

6 min read · Apr 18, 2024

33 4



Giovanny Espitia in Stackademic

Mastering RNNs for Stock Prediction: TensorFlow & Python Tutorial

Recurrent Neural Networks: A Comprehensive Guide to Time-Dependent Data AnalysisIntroduction

10 min read · Jan 9, 2024

36 1

Michael Keith in Towards Data Science

Five Practical Applications of the LSTM Model for Time Series, with Code

How to implement an advanced neural network model in several different time series contexts

11 min read · Sep 22, 2023

208 1

Iqra Muhammad

Time Series Forecasting using LSTM: An Introduction with Code Explanations

Time series forecasting plays an important role in stock market forecasting, finance, and weather prediction. Long Short-Term Memory (LSTM)...

3 min read · Nov 15, 2023

 6