Udacity Machine Learning Engineer Capstone Project: LSTM for Stock Price Prediction in SageMaker

Bo Ma

June 16, 2020

1 Problem Definition

1.1 Project Overview

Quantitative financial models have been used in investment firms and hedge funds to help understanding market behavior and developing profitable trading strategies. Stock market prediction tries to predict the future value of a stock or other financial instruments. However, the efficient-market hypothesis [1] claims that stock prices reflect all current information, and price changes are based on newly reveals information thus is unpredictable. Burton Malkiel, in his famous book A Random Walk Down Wall Street [2], states stock prices change is a "random" process and could not be accurately predicted by price history.

Given the unpredictability of the stock market, however, the goal of this project is to build a stock price predictor that estimates the stock price for any future dates based on past stock price patterns. Financial data is known to have a low signal to noise ratio, and it would be challenging to make accurate predictions just based on price history without considering any other sources such as fundamental, technical, sentiments, and factors data. Therefore, the focus is to get hands-on experience on the ML pipeline in SageMaker and explore the performance of recurrent neural network (RNN) for making time-series predictions.

Note that the adjusted close price is referred to as the stock price since the current price of a share of stock is constantly changing during an active trading day. A stock can be traded millions of times per day at different prices, and it would be overwhelming and in-practical to directly work on the raw price data. Therefore, it is convenient to aggregate them to open, high, low, close prices for conducting an analysis. In particular, the adjusted close price incorporates stock dividends or stock split into the close price that is commonly used in stock predictions.

1.2 Problem Statement

This is a regression problem where the model takes a sequence of closed prices in the past and produces the expected stock prices for a sequence of future dates specified by users. The Long Short Term Memory (LSTM) is used to solve the problem. I'm not expecting the model can make accurate stock price forecasts simply based on price histories. However, the LSTM model seems to be well fit the underlying problem, and I'd like to see how it performs.

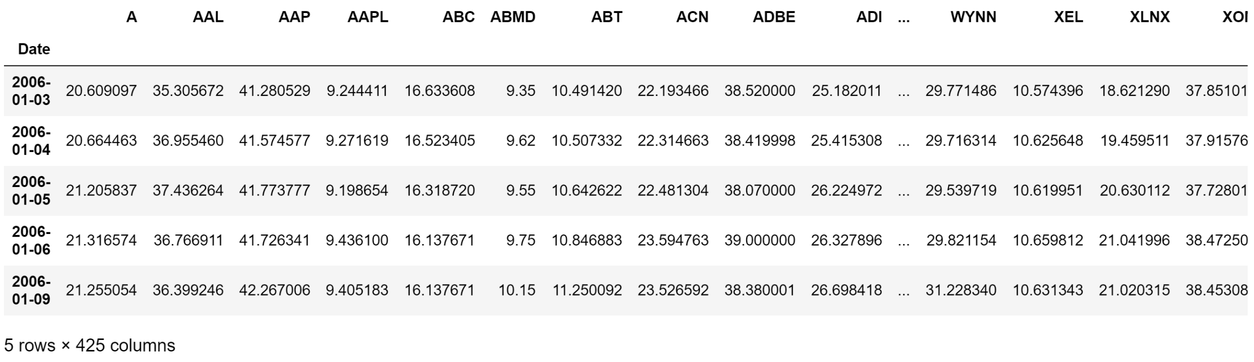
1.3 Metrics

Since the output is a sequence of expected prices for future dates, the mean squared error (MSE) between ground-truth and predictions is a natural choice to measure the performance of the model.

2 Analysis

2.1 Data Exploration

The SPX 500 Index is used as the stock universe. The tickers are obtained by scraping the wiki page ”List of S&P 500 companies” [3], and the stock price data is downloaded using an open-source package yfinance [4]. Users can select any number of stocks in the universe as inputs to the model. The time horizon spans from January 1, 2006 to June 9, 2020. After the data acquisition, data cleaning is performed to fill out missing values with last valid observation i.e., forward fill to avoid look-ahead bias. Then, tickers with all NA values are dropped. As a result, we end up with 425 stocks in our universe as shown in Fig.1, where the data has a daily sampling frequency.



	A	AAL	AAP	AAPL	ABC	ABMD	ABT	ACN	ADBE	ADI	...	WYNN	XEL	XLNX	XOI
Date															
2006-01-03	20.609097	35.305672	41.280529	9.244411	16.633608	9.35	10.491420	22.193466	38.520000	25.182011	...	29.771486	10.574396	18.621290	37.85101
2006-01-04	20.664463	36.955460	41.574577	9.271619	16.523405	9.62	10.507332	22.314663	38.419998	25.415308	...	29.716314	10.625648	19.459511	37.91576
2006-01-05	21.205837	37.436264	41.773777	9.198654	16.318720	9.55	10.642622	22.481304	38.070000	26.224972	...	29.539719	10.619951	20.630112	37.72801
2006-01-06	21.316574	36.766911	41.726341	9.436100	16.137671	9.75	10.846883	23.594763	39.000000	26.327896	...	29.821154	10.659812	21.041996	38.47250
2006-01-09	21.255054	36.399246	42.267006	9.405183	16.137671	10.15	11.250092	23.526592	38.380001	26.698418	...	31.228340	10.631343	21.020315	38.45308

5 rows × 425 columns

Figure 1: Sample stock data

2.2 Exploratory Visualization

Fig. 2 plots the adjusted close price of Microsoft and IBM. The price history is further divided into train (85%), validation (15%), and test (10%) datasets in chronological order. Firstly, different stocks exhibit different price history patterns especially when they are in different sectors with different characteristics. This was a concern for me to consider multiple stocks for a single model. The model would make a single prediction given a certain price history while the true price could vary per stocks even though they have similar price history patterns. However, a counter-argument would be different stocks probably have different price movement histories, and the model could be powerful enough to identify such subtleties via learning thus make correct predictions. Secondly, the price generally moves upward over time which indicates different feature scales at different time periods. Therefore,

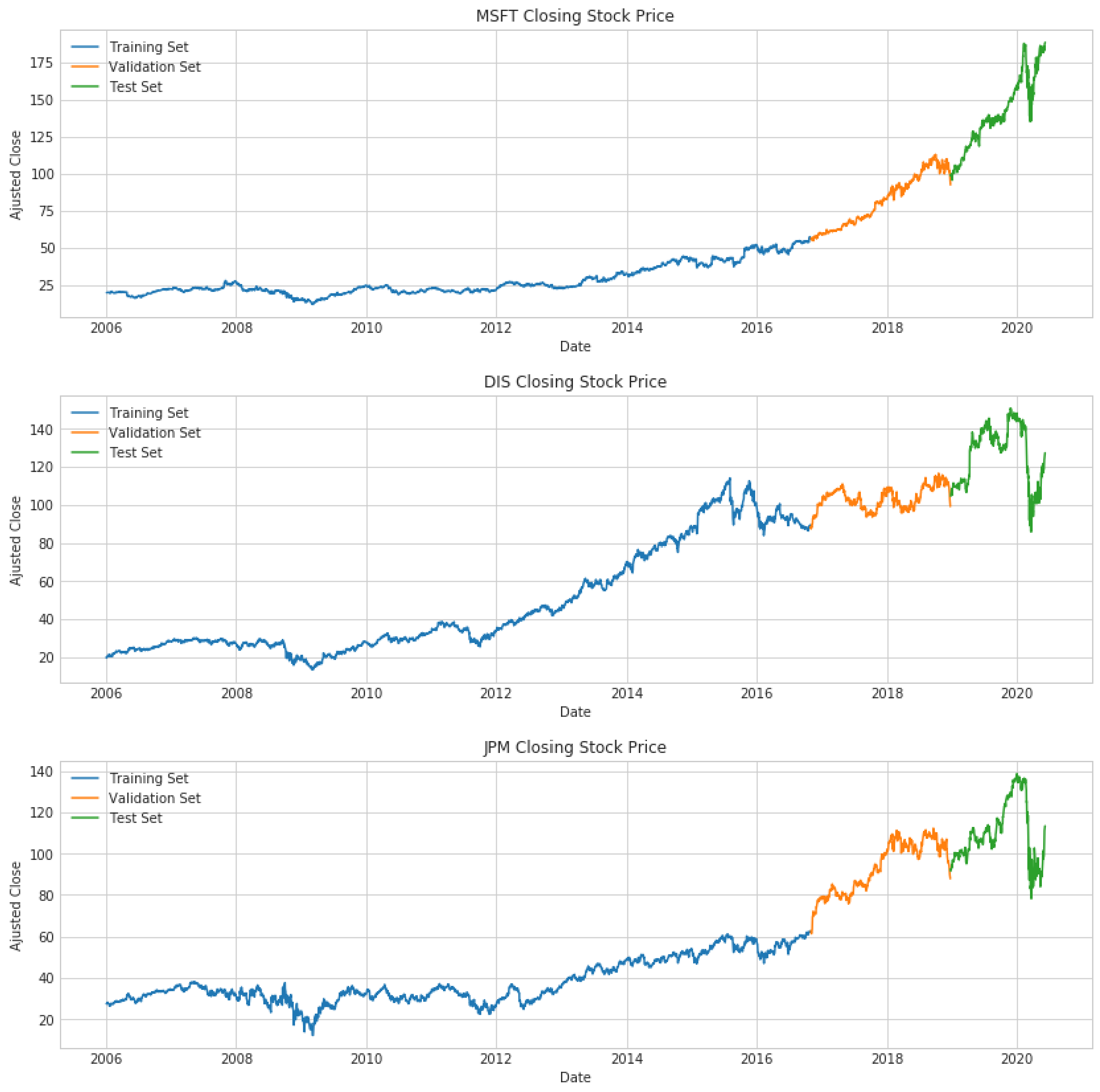


Figure 2: The visualization of sample stocks' price history.

feature normalization is needed; otherwise, a model trained on small scale data would not be able to predict large scale data in the future.

2.3 Algorithms and Techniques

The LSTM is an RNN architecture that is particularly effective for time-series predictions since it could incorporate important events with long lookback into predictions. The LSTM model will be trained for multiple tickers (specified by users) thus is expected to make predictions for any of these tickers.

2.4 Benchmark

Three simple models will be used to evaluate the LSTM model. The last value model makes predictions using the last observed model. The moving average model uses the average of past n values for predictions. The linear regression model fits a linear regression model to the previous n days and use the model for future predictions. The result will be compared with benchmark models both qualitatively (visualization) and quantitatively (MSE error).

3 Methodology

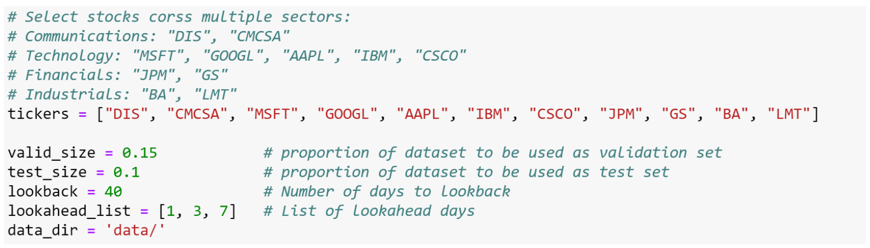
3.1 Data Preprocessing

After data cleaning (Sec 2.1), data preprocessing is needed to generate data in the format that is ready for training and testing. The module "datastore.py" is created which contains the class "DataStore" to handle preprocessing steps. The module takes the raw (cleaned) time-series data together with user specifications and generates data in a format that is ready to be fed into the model. The users' inputs are passed into the model in the "init" function with details documented in Fig. 3.

```
class DataStore:
    """ The class preprocesses the data, holds the raw data, and provides data accessors

        Attributes:
            train (pd.DataFrame)            : Raw train data (time series)
            valid (pd.DataFrame)            : Raw valid data (time series)
            test  (pd.DataFrame)            : Raw test data  (time series)

            train_dict (Dictionary)         : Stores the scaled data ready be to feed into the model for each ticker
                                              Usage:
                                                 train_dict['DIS']['scaler'] (MinMaxScaler(copy=True,
                                                                              feature_range=(0, 1))):
                                                     Scaler to transform (normalize) and inverse transform data

                                                 train_scaler['DIS']['yX'] (DataFrame): Data in model ready format
                                                     (Number of samples, len(lookahead_list) + lookback)
                                                     Each row is a sample.
                                                     First 'len(lookahead_list)' columns store (true) output data
                                                     for lookahead days of interest.
                                                     The rest 'lookback' columns store input data.
            valid_dict (Dictionary)         : Stores the scaled data ready be to feed into the model for each ticker
            test_dict (Dictionary)          : Stores the scaled data ready be to feed into the model for each ticker
    """
    def __init__(self, data: pd.DataFrame, tickers: list, valid_size: float, test_size: float, lookback: int, lookahead: int):
        """
            Parameters
            ----------
            data           : Stock universe
            tickers        : A list of stocks to be included in the model
            valid_size     : The proportion of the data used for the validation dataset
            test_size      : The proportion of the data used for the test dataset
            lookback       : Number of days to lookback
            lookahead_list : List of lookahead days. len(lookahead_list) determines the size of output sequence
        """
```

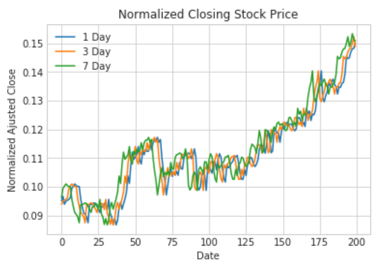
Figure 3: The DataStore class takes data and users inputs and generates model-ready data and provide data accessors.

The DataStore has accessor attributes "train_dict", "valid_dict", and "test_dict" to store all the normalized the data ready to be fed into the model for training and prediction. The data normalization is done by ticker to normalize data across the temporary dimension. The



```
# Select stocks corss multiple sectors:
# Communications: "DIS", "CMCSA"
# Technology: "MSFT", "GOOGL", "AAPL", "IBM", "CSCO"
# Financials: "JPM", "GS"
# Industrials: "BA", "LMT"
tickers = ["DIS", "CMCSA", "MSFT", "GOOGL", "AAPL", "IBM", "CSCO", "JPM", "GS", "BA", "LMT"]

valid_size = 0.15             # proportion of dataset to be used as validation set
test_size = 0.1               # proportion of dataset to be used as test set
lookback = 40                 # Number of days to lookback
lookahead_list = [1, 3, 7]    # List of lookahead days
data_dir = 'data/'
```

(a) Input



(b) Labels for lookahead 1, 3, and 7 days

Figure 4: Datesets and input.

scaler used for normalization is also stored for each ticker in the dictionary. The scaler can be used to convert between the raw and normalized data for different tickers.

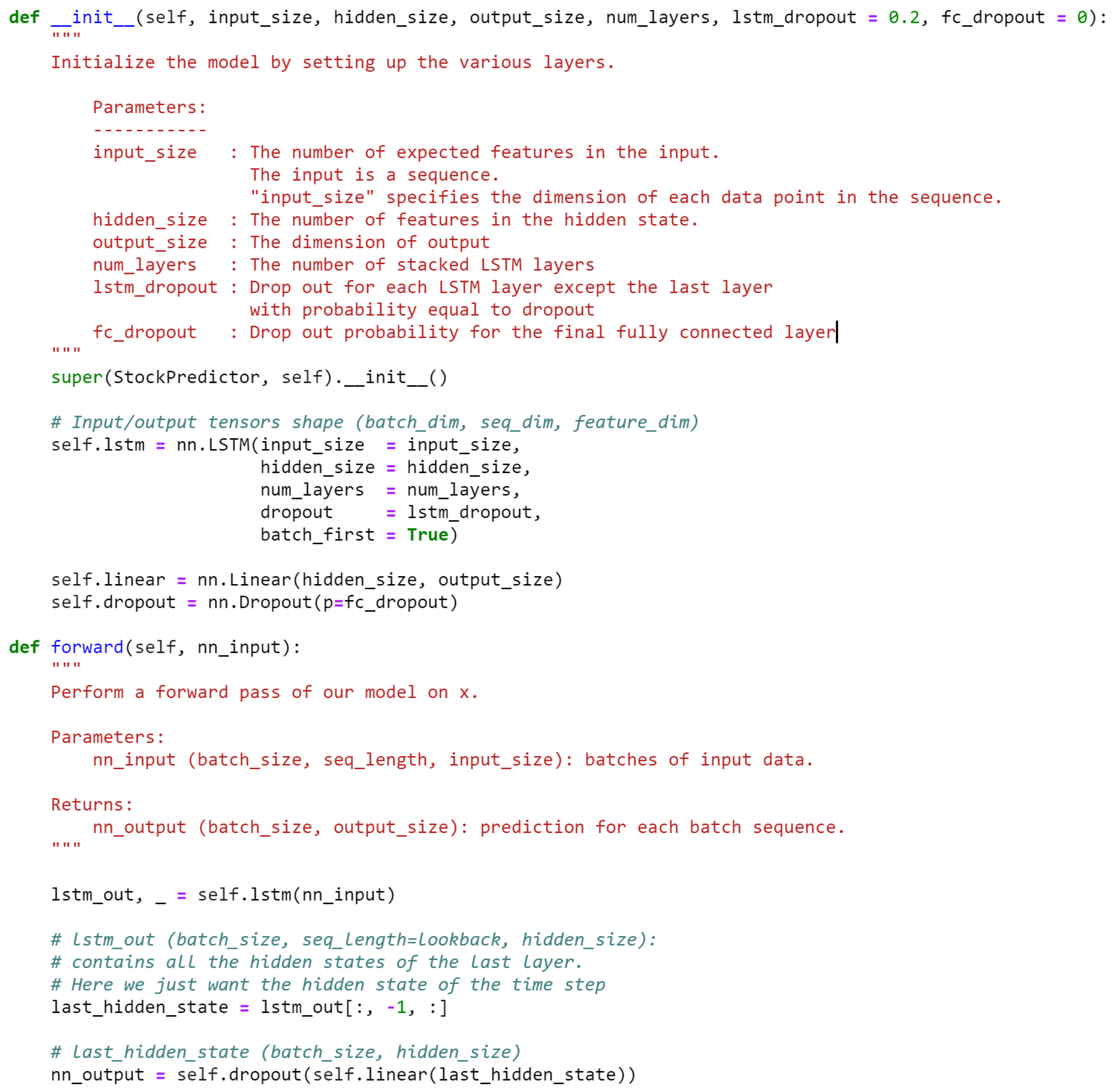
Since data is stored by tickers, the class provides "save_train_valid_to_csv" function to flatten data and store them into CSV files. As a result, each row represents a sample with first n columns represent labels and the rest of the columns represent features. For example, if the user set "lookback = 40" and "lookahead_list = [1, 3, 7]" meaning he wants to use data in day $1-40$ to predict stock prices in day $41, 43, 47$. Thus, the first 3 columns represent the prices in day $41, 43, 47$ while the rest columns represent data in day $1-40$. Fig 4b shows an example plot of expected prices or labels at day $1, 3, 7$.

3.2 Implementation

The (stacked) LSTM model is implemented using Pytorch. As shown in Fig 5, it is a standard LSTM model with stacked LSTM layers and a fully connected layer as the output layer. The last timestamp of the LSTM "hidden_state" is used as the input to the fully-connected layer for making predictions.

Fig. 4a shows the user inputs where a couple of tickers across different sectors are selected for training and predictions. The values for "lookback" and "lookahead_list" specifies that we want to use the price history in the past 40 days to make predictions for future $1, 3, 7$ days.

One of the complications during coding is the drop out of the final fully connected layer has a large impact on the training process. Fig. S1 shows the training and validation loss for various hyperparameters which will be discussed in detail later. Inspecting the "fc_dropout" column, we see the underlying hyperparameter has a strong influence on the train/validation loss pattern. Specifically, with 0 dropout, both train and validation losses are small indicating low bias/variance. However, with 0.1 probability dropout, the model exhibits high bias behavior. In general, high bias signals more complex models with longer training time. Since I do not expect increasing model complexity would help to solve the stock prediction problem, I still with the current model architecture. What's more, I set "lstm_dropout" and "fc_dropout" as hyperparameters for the SageMake hyperparametertunner to pick the best model.



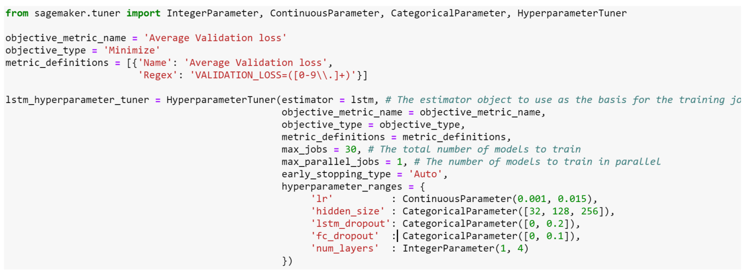
```
def __init__(self, input_size, hidden_size, output_size, num_layers, lstm_dropout = 0.2, fc_dropout = 0):
    """
    Initialize the model by setting up the various layers.

        Parameters:
        -----------
        input_size   : The number of expected features in the input.
                       The input is a sequence.
                       "input_size" specifies the dimension of each data point in the sequence.
        hidden_size  : The number of features in the hidden state.
        output_size  : The dimension of output
        num_layers   : The number of stacked LSTM layers
        lstm_dropout : Drop out for each LSTM layer except the last layer
                       with probability equal to dropout
        fc_dropout   : Drop out probability for the final fully connected layer
    """
    super(StockPredictor, self).__init__()

    # Input/output tensors shape (batch_dim, seq_dim, feature_dim)
    self.lstm = nn.LSTM(input_size  = input_size,
                        hidden_size = hidden_size,
                        num_layers  = num_layers,
                        dropout     = lstm_dropout,
                        batch_first = True)

    self.linear = nn.Linear(hidden_size, output_size)
    self.dropout = nn.Dropout(p=fc_dropout)

def forward(self, nn_input):
    """
    Perform a forward pass of our model on x.

    Parameters:
        nn_input (batch_size, seq_length, input_size): batches of input data.

    Returns:
        nn_output (batch_size, output_size): prediction for each batch sequence.
    """

    lstm_out, _ = self.lstm(nn_input)

    # lstm_out (batch_size, seq_length=lookback, hidden_size):
    # contains all the hidden states of the last layer.
    # Here we just want the hidden state of the time step
    last_hidden_state = lstm_out[:, -1, :]

    # last_hidden_state (batch_size, hidden_size)
    nn_output = self.dropout(self.linear(last_hidden_state))
```

Figure 5: The LSTM model.

3.3 Refinement

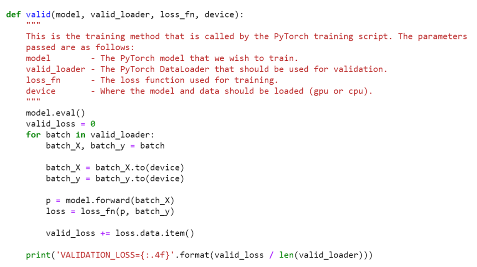
I leveraged SageMaker's HyperparameterTuner to help select the best hyperparameters. As shown in Fig. 6a, the hyperparameters include: "lr": the learning rate which is the most important hyperparameter in general. "hidden_size", "lstm_dropout", and "fc_dropout" is set to categorical parameters to regularize the search space. "num_layer" is a integer parameter from 1 to 4.

For the HyperparameterTuner to select the best model, we need to define an objective function. Fig. refobj shows the "valid" function defined in "train.py". The function is called after every training epoch to calculate validation loss. The validation loss is further printed



```python
from sagemaker.tuner import IntegerParameter, ContinuousParameter, CategoricalParameter, HyperparameterTuner

objective_metric_name = 'Average Validation loss'
objective_type = 'Minimize'
metric_definitions = [{'Name': 'Average Validation loss',
                       'Regex': 'VALIDATION_LOSS=([0-9\\.]+)'}]

lstm_hyperparameter_tuner = HyperparameterTuner(estimator = lstm, # The estimator object to use as the basis for the training jo
                                                objective_metric_name = objective_metric_name,
                                                objective_type = objective_type,
                                                metric_definitions = metric_definitions,
                                                max_jobs = 30, # The total number of models to train
                                                max_parallel_jobs = 1, # The number of models to train in parallel
                                                early_stopping_type = 'Auto',
                                                hyperparameter_ranges = {
                                                    'lr'          : ContinuousParameter(0.001, 0.015),
                                                    'hidden_size' : CategoricalParameter([32, 128, 256]),
                                                    'lstm_dropout': CategoricalParameter([0, 0.2]),
                                                    'fc_dropout'  : CategoricalParameter([0, 0.1]),
                                                    'num_layers'  : IntegerParameter(1, 4)
                                                })
```

(a) Hyperparameter tuning



```python
def valid(model, valid_loader, loss_fn, device):
    """
    This is the training method that is called by the PyTorch training script. The parameters
    passed are as follows:
    model        - The PyTorch model that we wish to train.
    valid_loader - The PyTorch DataLoader that should be used for validation.
    loss_fn      - The loss function used for training.
    device       - Where the model and data should be loaded (gpu or cpu).
    """
    model.eval()
    valid_loss = 0
    for batch in valid_loader:
        batch_X, batch_y = batch

        batch_X = batch_X.to(device)
        batch_y = batch_y.to(device)

        p = model.forward(batch_X)
        loss = loss_fn(p, batch_y)

        valid_loss += loss.data.item()

    print('VALIDATION_LOSS={:.4f}'.format(valid_loss / len(valid_loader)))
```

(b) Custom objective function

Figure 6: Datesets and input.

in the log. In Fig. refhyperprame, the ”metric_definition” object picked up the validation loss printed in the log as the evaluation metric for different models.

Fig. S1 shows the training and validation loss for various hyperparameters sorted by the ”Validation Loss” column. We can see simple networks with small ”hidden_size” and ”num_layers” are generally performs better. The top three models have almost the same validation error. Model 14 is picked by SageMaker as the best model.

Table S1: The result of 17 model runs.

Model	HyperParameters					Early Stopping	Train Loss	Validation Loss
	lr	hidden_size	lstm_dropout	fc_dropout	num_layers			
3	0.003	128	0	0	1	False	0.0007	0.0028
14	0.002	32	0.2	0	3	False	0.0011	0.0028
9	0.003	32	0	0	2	False	0.0004	0.0028
6	0.006	128	0	0	2	False	0.0011	0.0030
5	0.002	128	0.2	0	4	False	0.0011	0.0031
2	0.009	128	0.2	0	1	False	0.0014	0.0031
12	0.001	32	0	0.1	1	False	0.0256	0.0063
10	0.004	256	0.2	0.1	1	True	0.0257	0.0065
17	0.004	128	0	0.1	2	True	0.0258	0.0065
16	0.001	128	0	0.1	2	True	0.0258	0.0066
8	0.001	128	0.2	0.1	2	False	0.0258	0.0068
15	0.001	256	0	0.1	3	True	0.0260	0.0068
11	0.003	256	0.2	0	2	False	0.005	0.0284
4	0.014	256	0	0	4	True	0.275	0.0593
1	0.008	32	0.2	0.1	4	False	0.093	0.0839
13	0.01	256	0	0	2	True	0.1727	0.1247
7	0.014	128	0.2	0	4	True	0.129	0.1982

4 Results

4.1 Model Evaluation and Validation

Fig. 7 shows the resulting predictions for various stocks with 1 day lookahead. Note that the training MSEs are computed based on normalized data while the MSEs in Fig. 7 are calculated based on real stock prices. We can see our model does a pretty good job as the predictions generally follow the real price movements. Wait a minute, this model can predict stock prices! I'm going to be a billionaire! Alright, before I invest all my money, let's do a sanity check for lookahead 7 days. Hmm..., Fig. 8 is not that pretty. It seems the predictions are lagged. It seems our estimator is kind of following the previous day or it works as a kind of moving average style. Therefore, it makes similar predictions regardless of how many look ahead days are of interest. Fair enough, I would be surprised if any estimator can accurately predict stock prices in weeks or months without knowing the price movement in between.

4.2 Justification

Fig. 9a compared the average MSEs over all tickers of the LSTM model with simple benchmark models: last Value, moving average (over lookback days), and linear regression. The LSTM model has comparable MSEs to the last value and linear regression model. Fig. 9b plots the predictions generated by these models for lookahead 1 day. Except for the moving average, all other models are indistinguishable. This further verifies that the LSTM model is just following the recent price movement.

In conclusion, the final results are pretty bad and our model can not predict stock prices accurately, which, on the other hand, is expected. After "hard" training, the LSTM model still performs similarly to the simplest benchmark models which far from satisfactory. However, my goal is achieved: I learned a lot of the ML workflow in SageMaker, especially gain valuable experiences in hyperparameter tuning. What's more, now I believe Burton Malkiel that the stock market is unpredictable - at least not by simple models :).

References

[1] Malkiel, Burton G. "The efficient market hypothesis and its critics." Journal of economic perspectives 17.1 (2003): 59-82.

[2] Malkiel, Burton Gordon. A random walk down Wall Street: including a life-cycle guide to personal investing. WW Norton Company, 1999.

[3] *List of S&P 500 companies*, available at `https://en.wikipedia.org/wiki/List_of_S%26P_500_companies`.

[4] Yahoo! Finance market data downloader, available at `https://github.com/ranaroussi/yfinance`.

[5] *Stock market prediction*, available at `https://en.wikipedia.org/wiki/Stock_market_prediction`.

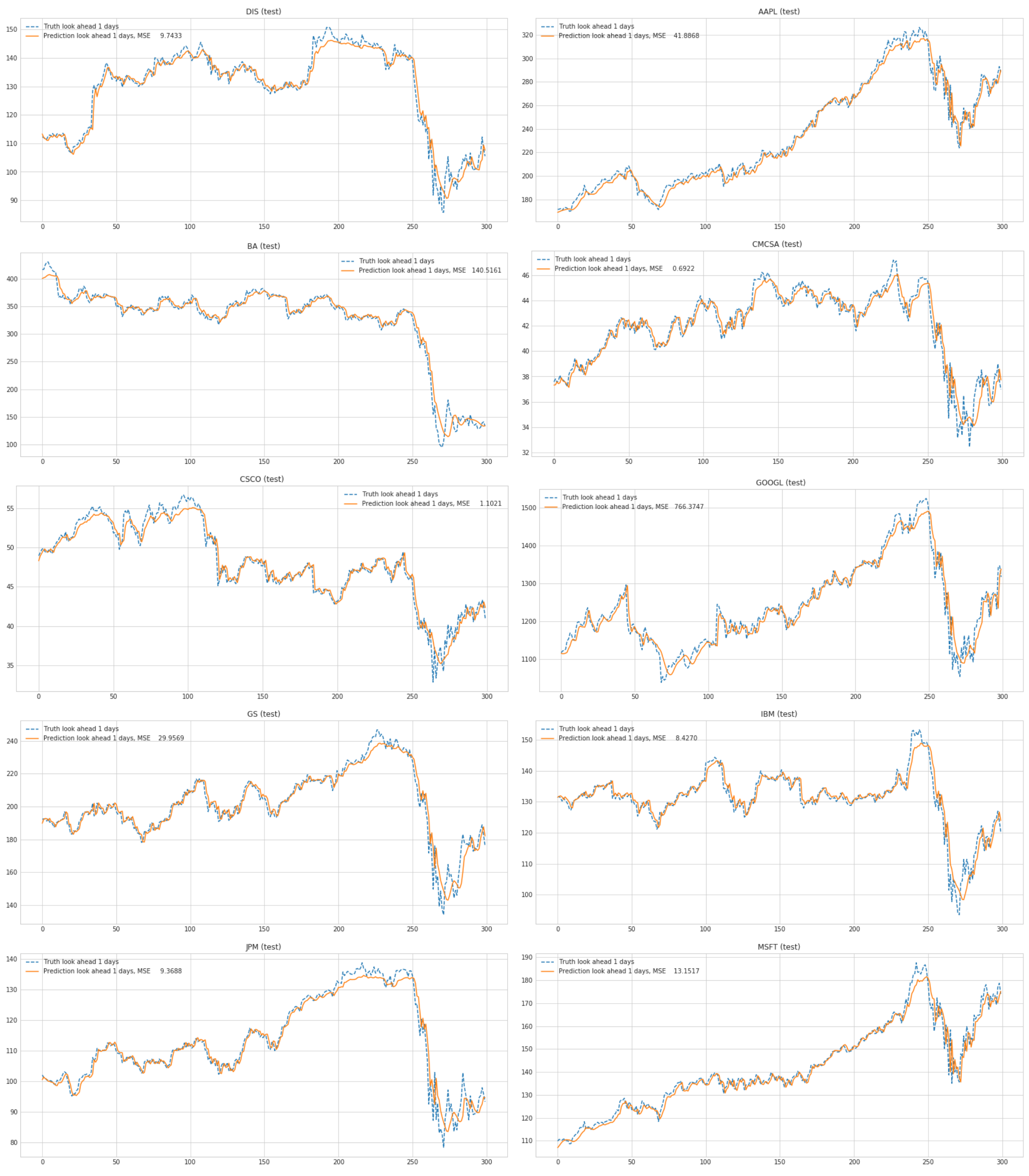
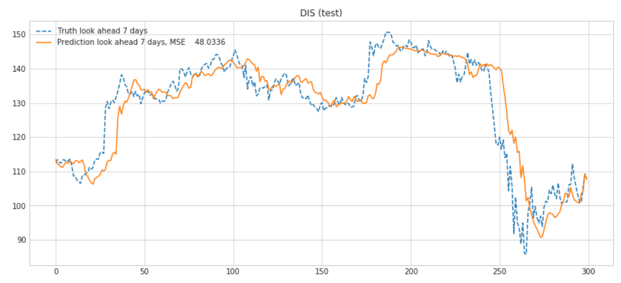
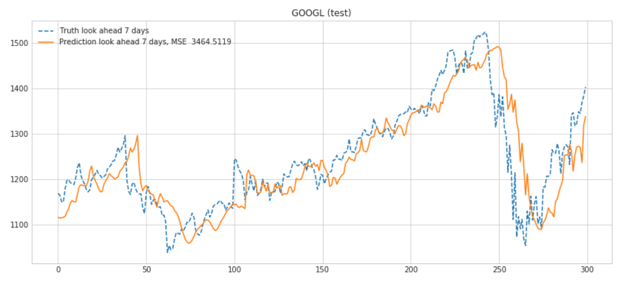


Figure 7: Visualization and MSE for test dataset with 1 day look ahead

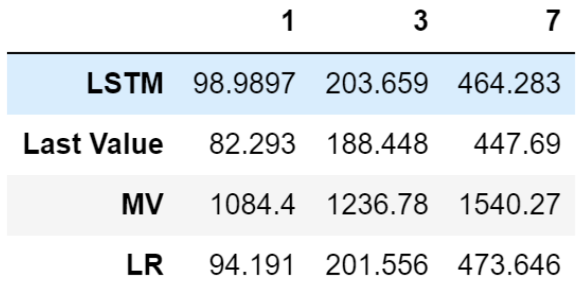


(a)



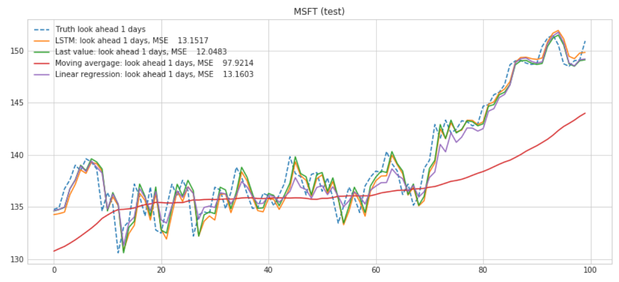
(b)

Figure 8: Visualization and MSE for test dataset with 7 day look ahead



	1	3	7
LSTM	98.9897	203.659	464.283
Last Value	82.293	188.448	447.69
MV	1084.4	1236.78	1540.27
LR	94.191	201.556	473.646

(a)



(b)

Figure 9: The comparison with benchmark models using MSE.