# Stock Price Prediction using a LSTM Model and Transfer Learning

by Seita Yoshifusa, Andreas Koni, Jimmy Hou

## Project Objective

As the ties between various nations become stronger with increasing international exchanges, the travel rate also undergoes an increase. When we consider traveling somewhere physically distant, the first related industry that comes to our mind is the airline industry. In today's market, we have a near duopoly in passenger aircraft manufacturing between the American Boeing and the European Airbus.

Given this economic circumstance in the plane-making industry, we were interested in seeing if we could construct a generalized stock price model that can accurately track and forecast future stock prices specifically for companies in the plane-making industry.

In achieving our objective, our project comprises two main components:

1. the construction of our own LSTM model
2. the implementation of transfer learning based on the pre-trained LSTM model.

## Data Preprocessing

We gathered our two datasets for the stock price histories of Boeing Company and Airbus SE from Yahoo Finance. We first created and trained our model on the Boeing Company dataset and then performed transfer learning on the Airbus SE dataset. For both datasets, we did the same data preparation process as follows.

Before downloading the .csv file from Yahoo Finance, we had to choose a specific time frame to take the data. The dataset tracked historical data back at least 20 years, but we narrowed down the data to a ten-year timeframe with daily updates from March 5, 2014, to March 5, 2024. We then read the .csv file into a pandas DataFrame. The dataset initially had six columns: Date, Open, High, Low, Close, Adj Close, and Volume. For our model, we were only interested in the Date and Close columns, so we filtered out the other columns and also got rid of any rows that had NaN values for the Close price.

Next, we prepared our data to easily split into the target variable and the predictor matrix arrays. To prepare our dataset, we wanted to look at the prices for each day within the previous week of the close price we wanted to predict. Thus, from our original dataset with two columns, we added seven more where each column added on was the number of previous days to look back at. The formatted dataset appeared as follows:

| Date | Close | Close(t-1) | Close(t-2) | Close(t-3) | Close(t-4) | Close(t-5) | Close(t-6) | Close(t-7) |
|---|---|---|---|---|---|---|---|---|
| **2014-03-14** | 123.110001 | 121.889999 | 124.430000 | 125.669998 | 126.889999 | 128.539993 | 128.860001 | 128.789993 |
| **2014-03-17** | 125.419998 | 123.110001 | 121.889999 | 124.430000 | 125.669998 | 126.889999 | 128.539993 | 128.860001 |
| **2014-03-18** | 124.040001 | 125.419998 | 123.110001 | 121.889999 | 124.430000 | 125.669998 | 126.889999 | 128.539993 |
| **2014-03-19** | 122.239998 | 124.040001 | 125.419998 | 123.110001 | 121.889999 | 124.430000 | 125.669998 | 126.889999 |
| **2014-03-20** | 123.730003 | 122.239998 | 124.040001 | 125.419998 | 123.110001 | 121.889999 | 124.430000 | 125.669998 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... |
| **2024-02-28** | 207.000000 | 201.399994 | 200.539993 | 200.830002 | 201.500000 | 201.570007 | 203.369995 | 203.889999 |
| **2024-02-29** | 203.720001 | 207.000000 | 201.399994 | 200.539993 | 200.830002 | 201.500000 | 201.570007 | 203.369995 |
| **2024-03-01** | 200.000000 | 203.720001 | 207.000000 | 201.399994 | 200.539993 | 200.830002 | 201.500000 | 201.570007 |
| **2024-03-04** | 200.539993 | 200.000000 | 203.720001 | 207.000000 | 201.399994 | 200.539993 | 200.830002 | 201.500000 |
| **2024-03-05** | 201.139999 | 200.539993 | 200.000000 | 203.720001 | 207.000000 | 201.399994 | 200.539993 | 200.830002 |

2511 rows × 8 columns

*The columns labeled as 'Close (t - k)' represent the close prices on the k-th previous day.*

After formatting the data, we randomly split our shuffled data into 80% for our training set and 20% for our testing set. Next, we standardized the training and

testing set separately using the `MinMaxScaler()` from the sci-kit learn package on Python. We chose this standardization technique because it preserves the shape of the data while making the computation easier when running the data through our model. We also performed the standardization over the feature range from -1 to 1 to track fluctuations around zero, which would help track fluctuations in the close prices. After standardizing the sets separately, we split each of them into the target variables and predictor matrices. The target variable vector was simply the close prices column from the set. The predictor matrix was the rest of the columns representing the close prices of the days to look back on. The shapes of the arrays looked as follows:

| Array | Shape |
|---|---|
| X_train | (2008, 7) |
| X_test | (503, 7) |
| y_train | (2008, ) |
| y_test | (503, ) |

In order to input these arrays into our LSTM model, we had to reshape the arrays by adding an extra dimension to them. For example, the shape of the `X_train` and `y_train` arrays will change from (2008, 7) to (2008, 7, 1) and (2008, ) to (2008, 1). The same follows for the `X_test` and `y_test` arrays. The extra dimension specifies the number of features input into the LSTM, where in our case we were only concerned with the close price of the stock, so the third dimension is just one.

Now, for the final stage of data preparation, we prepared the data for proper use in our Pytorch LSTM model. First, we converted our arrays to torch tensors, which allowed the gradients of the arrays to be easily tracked and updated with the arrays. We also defined our own custom TimeSeriesDataset class, which was inherited from the Pytorch Dataset package. We created class objects to allow easier access to the data using the Date column. So, we created one object for the training data and one object for the testing data, which paired the target variables and predictor matrices using the Date column. Next, we used the DataLoader package from Pytorch to process the new TimeSeriesDataset objects into batches of the designated size. For the time being, we ran the DataLoaders with a batch

size of 16, but we will see later on that we ran Grid Search to optimize this parameter.

Finally, our data was preprocessed and ready to run through our Pytorch LSTM model.

## Construction of LSTM Model

Once the data was preprocessed, we needed to construct our own LSTM, or long-short term memory, model. LSTM is essentially a type of Recurrent Neural Network architecture, and it is capable of selectively remembering or forgetting information over time by leveraging a series of “gates” that have assigned weights and undergo specific activation functions to regulate information. This sophisticated type of neural network is suitable for learning and retaining long-term dependencies in sequential data, and hence, an ideal candidate for constructing our stock price prediction model.

In order to ensure that we construct our LSTM model in the most optimal fashion, we also implemented a method known as Grid Search. Grid Search is essentially an exhaustive hyper-parameter tuning technique that finds the best combination of hyper-parameters for a given model by iterating through all possible combinations of specified hyper-parameter values. The optimality of the hyper-parameter values under Grid Search was selected using Mean Squared Error(MSE) as the loss function, which will be explained more in-depth in this section later on.

The construction of the LSTM model itself is relatively basic. As shown below, we define a class object that takes a single input ( `input_size` ), a specified number of hidden layers of the LSTM network ( `hidden_size` ), a specified number of stacked layers to enhance feature extraction and increase model capacity, and a dropout and a weight decay argument as a regularization method to prevent overfitting. The forward pass definition embedded inside the LSTM class allows the input data `x` to flow through the fully connected network with the specified parameters to return the final output of the model.

```python
class LSTM(nn.Module):
    """
    LSTM class defines the construction of the PyTorch LSTM model.

    Args:
    - input_size: Number of neurons in the input layer.
    - hidden_size: Number of neurons in each hidden layer.
    - num_stacked_layers: Number of hidden layers.
    - dropout: Dropout rate (default: 0.0).
    - weight_decay: Weight decay (default: 0.0).

    Attributes:
    - hidden_size: Number of neurons in each hidden layer.
    - num_stacked_layers: Number of hidden layers.
    - lstm: LSTM layer.
    - fc: Fully connected output layer.
    - dropout: Dropout layer if dropout rate > 0, otherwise None.
    - weight_decay: Weight decay.
    """

    def __init__(self, input_size, hidden_size, num_stacked_layers, dropout=0.0, weight_decay=0.0):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers

        # LSTM layer
        self.lstm = nn.LSTM(input_size, hidden_size, num_stacked_layers, batch_first=True)

        # Fully connected output layer
        self.fc = nn.Linear(hidden_size, 1)

        # Dropout layer if dropout rate > 0
        self.dropout = nn.Dropout(dropout) if dropout > 0.0 else None

        # Weight decay
        self.weight_decay = weight_decay

    def forward(self, x):
        """
        Forward pass through the LSTM model.

        Args:
        - x: Input data.

        Returns:
        - out: Output from the model.
        """
        batch_size = x.size(0)
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device)
        c0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device)

        # Forward pass through the LSTM layer
        out, _ = self.lstm(x, (h0, c0))

        # Extract the output from the last time step
        out = self.fc(out[:, -1, :])

        # Apply dropout if specified
        if self.dropout:
            out = self.dropout(out)

        return out
```

## Training

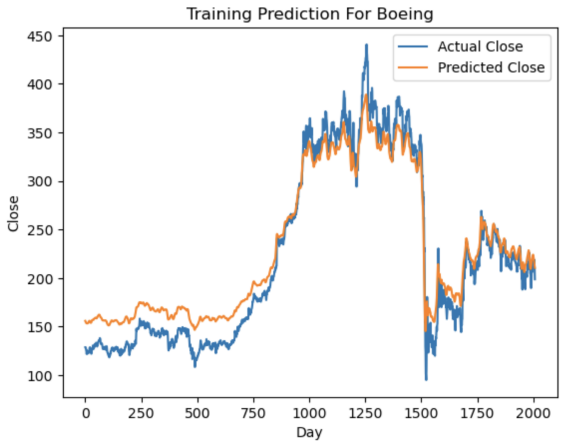
As the model groundwork was constructed, we assigned random hyper-parameters to initialize the model. We then applied Grid Search under a dictionary of parameters that we defined in order to find the most optimal parameters for the LSTM model using MSE as the loss function and Adam as the optimizer as these are generally regarded as the best choices respectively when dealing with handling stock prices. In our case, the Grid Search results indicated that our hyper-parameters should have batch size = 32, learning rate = 0.001, number of epochs = 6, hidden size = 32, input size = 1, and number of stacked layers = 1.

Once we extract the optimal parameters from Grid Search, we manually input those values as the new hyper-parameters for our LSTM model. Again, as we are

dealing with stock prices, we decided to use the MSE as the loss function and Adam as the optimizer. In order to visualize the training process under the updated model, we printed out and confirmed that training loss values and validation loss values properly decreased as we iterated through the optimal number of epochs specified by the Grid Search.

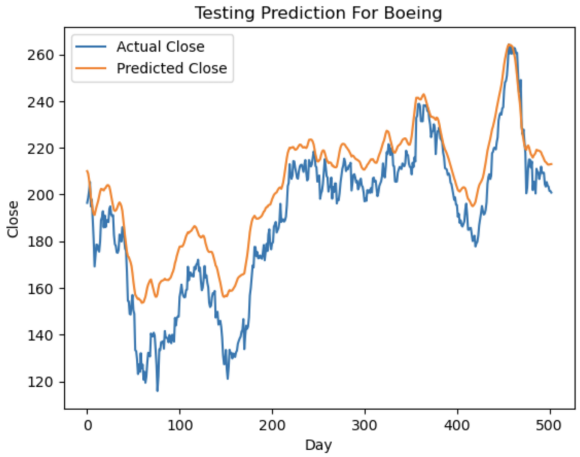
After we trained our updated LSTM model under the Boeing dataset, we extracted the target and prediction values to compare how our model fit the dataset. One thing to note during this process is that since we used the MinMaxScaler to transform our output values to be between -1 and 1, we had to call the `scaler.inverse_transform` method to revert the scaled values to its original form.

Finally, we visualized how well the model followed the actual stock prices for our training dataset, (Boeing stock prices in our case) and saw that the model was successful in following the general shape of the training target values without extreme overfitting as shown on the right.

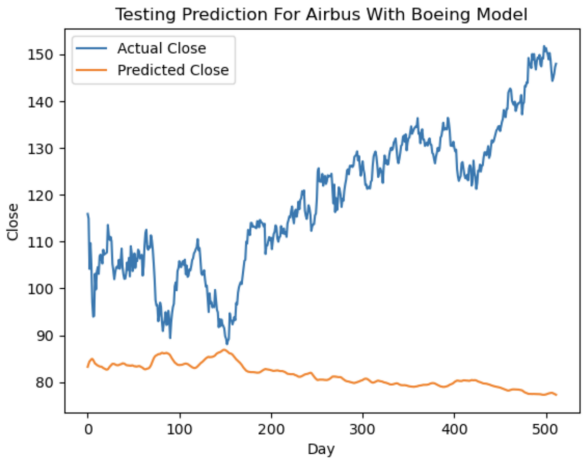



# Testing

After we finalized our training process as explained in the above section, we decided to see if the constructed LSTM model could accurately predict stock prices for Boeing on the test dataset. Given how well the trained model fit the training dataset, the results were within our expectations. The model performed very well in the sense that it outputted prediction values that were close to the actual target values, as illustrated on the graph on the right.




However, we have not yet constructed a “generalized” model. If we use our trained model to predict stock prices for the Airbus company, we clearly see that the model does not perform well with completely wrong predictions. The reason is simple. Our original LSTM model was built by learning features necessary to track stock prices for Boeing; hence, the model did not learn any features necessary to predict the stock prices for the Airbus company.



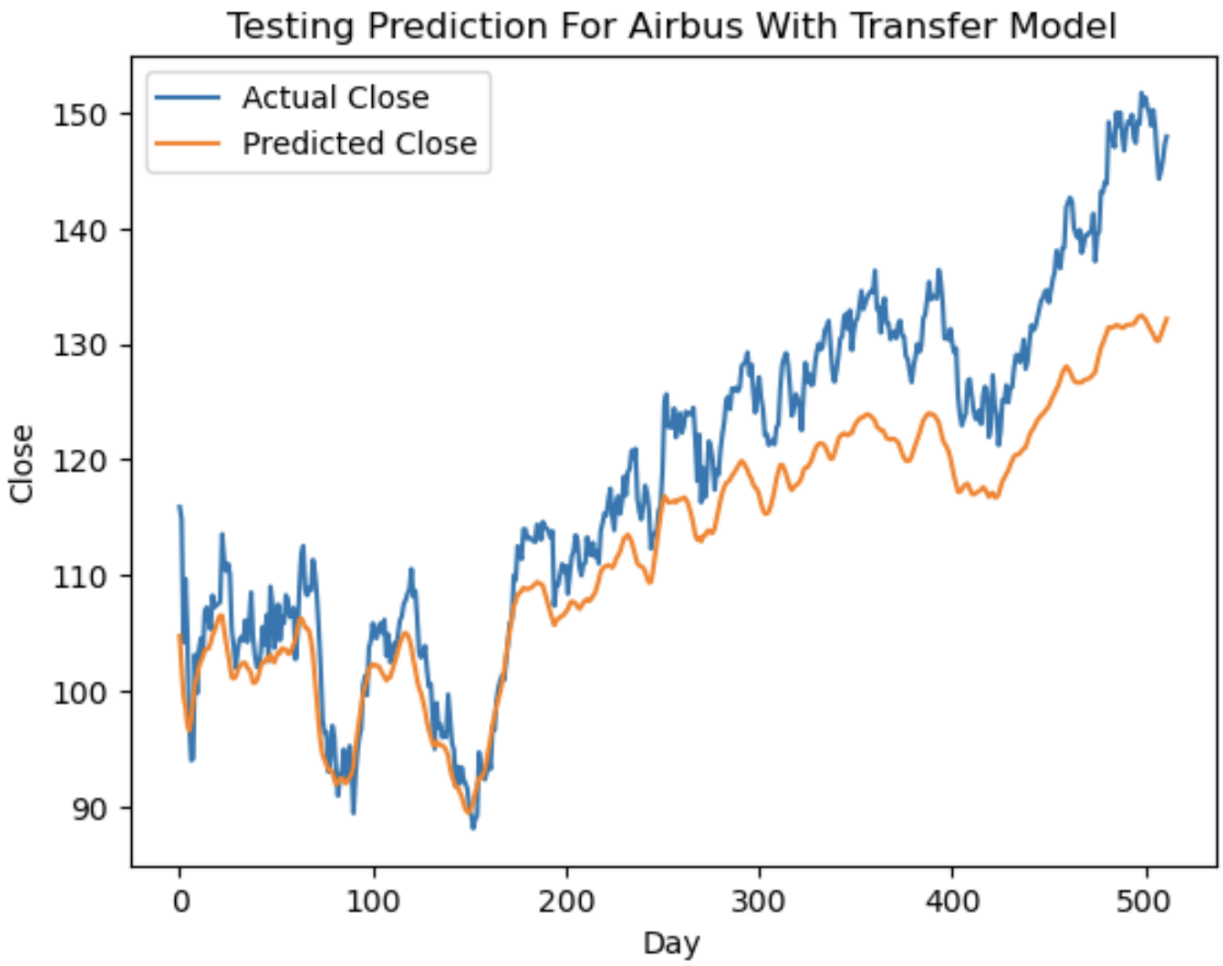

## Transfer Learning

In order to construct a truly “generalized” model for predicting stock prices for companies in the airline industry, we decided to leverage a method called transfer learning. Transfer learning, as the name suggests, essentially refers to a re-use method of a pre-trained model on a new problem. It is a popular method given that

this method saves computing time and allows for better performance of neural network models. For this project, we treated our original LSTM model as the pre-trained model in the transfer learning process so that our newly constructed model can learn features necessary to accurately predict stock prices for the Airbus company while also retaining information needed to predict stock prices for Boeing.

Similar to its original LSTM model counterpart, the construction of the new model leveraging transfer learning is very simple. To begin with, the parameters required for transfer learning are initialized to take in `input_size` (the size of the input features), `pretrained_model` (the reference model that will serve as the baseline for transfer learning), and `num_classes` (expected number of outputs). In our case, we store the original LSTM model that we constructed in a separate dummy variable and feed that dummy variable into the construction of our transfer learning model. Under the forward pass method of the transfer learning class, we define our input data to go through our pre-trained LSTM model along with a linear layer to ensure that it adapts to the new task.

After the construction of our transfer learning model using a pre-trained LSTM model, the rest of the process is essentially a repetition of the previous steps: we first perform data preprocessing for Airbus stock price data by performing train-test split and using MinMaxScaler to scale the values to save computing time, we train the transfer learning model using the Airbus stock price data as the training set, and finally, confirmation that the newly trained model fits the training set, we use that model to see if it accurately predicts the stock price for Airbus. Furthermore, we also tested to see if the new model constructed using transfer learning successfully became a generalized model by using the model to see if it accurately predicted stock prices for Boeing as well. Ultimately, as shown in the visualizations below, we saw that with the use of transfer learning, we were able to construct a generalized stock price prediction model specifically for the airline industry.




## Conclusion

Overall, we clearly saw that our initial LSTM model was more limited and restrictive given that it was only capable of tracking stock prices for Boeing, and gave completely incorrect predictions for the Airbus stock price data. However, upon implementing transfer learning, the original model was able to learn new features necessary to accurately predict stock prices for Airbus while also retaining the features of the Boeing data. Hence, the ultimate model that we constructed became capable of predicting stock prices for the two companies that dictate the airline industry in today's market.

## Difficulties

One of the first difficulties with the project was the use of a Pytorch LSTM model. Tensorflow is generally easier to construct and run, whereas Pytorch is much

more customizable and thus a little more meticulous in its construction. We had to have a much better understanding of how an LSTM works and the different hyper-parameters and parameters that go into it. In the end, we were able to construct our own with a good sense and understanding of Pytorch and its benefits. Another difficulty was running the grid search on our original model. This task could really only be done in a sufficient amount of time with the limited use of a GPU through Google Colab. With the use of a CPU, the computation time takes much longer. This also affects the amount of datasets that we can run the model on as well.

## Things Learn From This Project

Through my stock price prediction project, I gained hands-on experience in several crucial techniques and tools in the realm of machine learning. Leveraging Long Short-Term Memory (LSTM) models enabled me to effectively capture and utilize sequential data patterns for forecasting stock prices. Additionally, employing Grid Search and transfer learning methodologies allowed me to optimize model performance and adapt pre-trained models to new datasets, respectively. The utilization of MinMaxScaler facilitated the preprocessing of numerical data, ensuring it fell within a desired range for improved model performance. Moreover, working with TimeSeriesDataset and skorch enabled me to efficiently handle time-series data and implement neural network regression. Lastly, I became adept at utilizing GridSearchCV and understanding metrics such as neg_mean_squared_error to fine-tune model hyperparameters and assess model performance.

## Open Question

One problem of our project could be the generalizability of the developed model across different market conditions. While the model may perform well during training and testing phases, its efficacy in real-world scenarios, especially in volatile markets or during economic downturns, remains uncertain. Investigating how the model adapts to changing market dynamics, potential biases, and unexpected events could provide valuable insights into its robustness and reliability for practical applications.