Deep Learning Analysis

Tatev Karen Aslanyan

Recurrent Neural Networks Case Study

Predicting the Stock Price of Google

---

1 Introduction

In Quantitative Finance, predicting the stock price with high accuracy is a very important and difficult task. Although, it is impossible to exactly estimate the stock prices, it is possible to make accurate stock price predictions by using the past stock prices, which is also the theory behind Brownian Motions. This prediction is done by following closely the *downward* and *upward trends* in the past stock price development and use this as an indicator of future prices. In this analysis we will use Deep Learning, more specifically Long Short-Term Memory model (LSTM) to predict the price of the Google stock. LSTM is more sophisticated version of RNN which addressed the *Vanishing Gradient Problem* that RNNs often suffer from. This project is based on the past Google stock prices of the last 5 years corresponding the time period of 2016-2020. Our goal is to use this data to predict the upward or downward trends in the stock price of Google on January 2021.

2 Data

In this analysis we use Google's stock price data from the period of 2012-2016. More specifically, the data contains stock prices from January 2016 until December of 2020. That is a 5 year stock price data. This will act as a training data that will be used to train the LSTM on, whereas the stock price data of January 2021 will be used as a test data. Both these data sets contain the following data fields incorporated in Table 1. In

Variable	**Description**
Date	The date of the reported stock price
Open	The stock price at the beginning of the financial day
High	The highest price that the stock reached at that date
Low	The lowest price that the stock reached at that date
Close	The stock price at the end of the financial day
Volume	The number of shares transacted on that day

Table 1: Google Financial Data Variables

this analysis we will use the *Open* prices, the prices reported at the beginning of the trading day. Open stock prices development of Google stock is described by Figure 1.

2.1 Data Preprocessing

Firstly, we perform feature scaling on the data. To do this we can either use *Standardisation* or *Normalization*.

$$X_{standardized} = \frac{X - \mu_x}{\sigma_x} \qquad X_{noruamalized} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

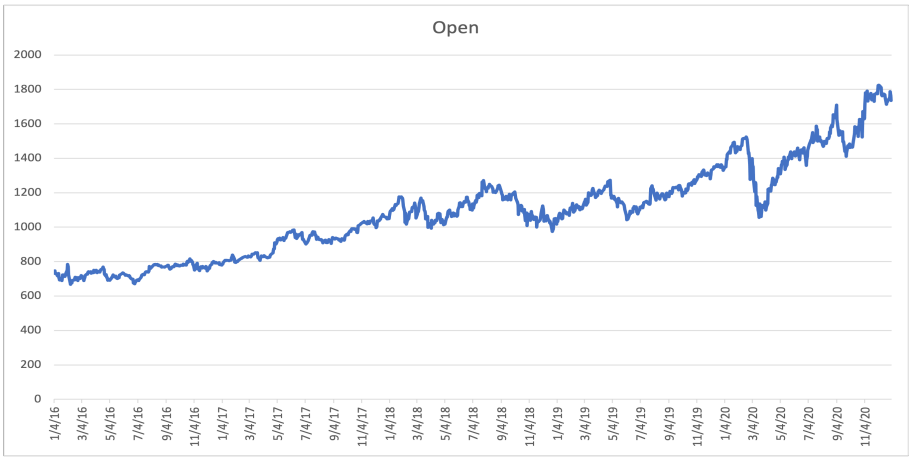


Figure 1: Google Stock Price During the 2016-2020 Time Period

where μ_x represents the mean of vector X and the σ_x represents the standard variation of vector X. Furthermore, X_{min} and X_{min} refer to the minimum and maximum values of vector X, respectively. In case of Normalization, as the denominator will always be larger than the nominator, the output value will always be a number between 0 and 1. Given that our methodology is based on LSTMs that include Sigmoid function, we will use normalization rather than standardization as a feature scaling method. This transformation can be done using the *MinMaxScaler* from the *Scikit-Learn* Python library.

A month consists of 20 trading days, hence we assume that 3 monthly data structure is appropriate, that is, we choose 60 as the number of timesteps. So, per observations, the X_{train} will contain the stock prices of the last 60 days and the Y_{train} will be the stock price of that day and this holds for all time periods (for all trading days in our data starting from the 60^{th} day of our data) such that we end up with a *pandas* data frame that in the first column the Y_{train} where each row is one data point from the original stock price data and starting from the second column, per row, we have the past 60 stock prices corresponding to that data. Figure 2 visualizes this process.

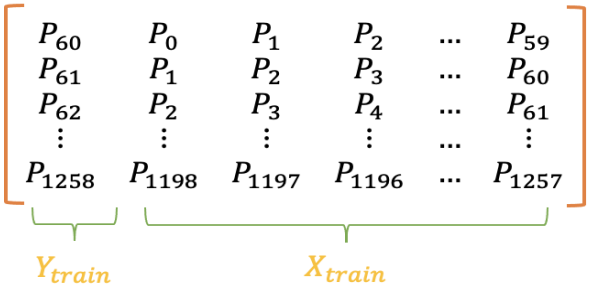


Figure 2: Stock Price Data Transformation

3 Methodology

The entire idea behind Deep Learning is to mimic the functional capabilities of human brain. One of the things that makes ANN very special, beside the back propagation, are the stored weights that keep the information as long as required until they get optimally updated in the next epoch. More specifically, ANNs can learn from prior observations thanks to the weights that are saved in each training epoch and those weights can be described as a *long-term memory*. When looking at a structure of a brain, then the *Temporal Lobe* is the part of the brain that is responsible for the long-term memory. Hence, ANN can be associated with this part of the brain, the Temporal Lobe.
On the other hand, CNNs are for image recognition and visual interpretation. So, it can be associated with the *Occipital Lobe* which is the visual processing center of our brain containing most of the anatomical region of the visual cortex.
Finally, the *Frontal Lobe* which is responsible for our *short-term memory* and has many other functionalities such as skeletal movement, ocular movement, speech control, and the expression of emotions. Recurrent Neural Networks DBLP can be associated with this part of the brain, the *Frontal Lobe*.
In case of RNN's the relationship between the inputs and the outputs is not one-to-one

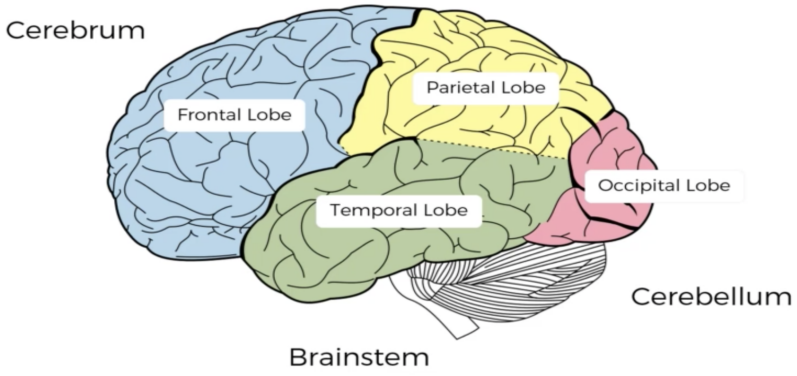


Figure 3: Human Brain's Structure

but one-to-many (one input and multiple output) or many-to-one (multiple inputs and single output). In the left hand-side of Figure 4 one can see an example of a *many-to-one* relationship where the model takes as an input an image of a dog and learns about it after which it provides multiple outputs (words) describing that picture. Additionally, in the right hand-side of the same figure one can see RNN model with *many-to-one* where a review consisting of multiple inputs (words) is classified as positive or negative using semantic analysis.

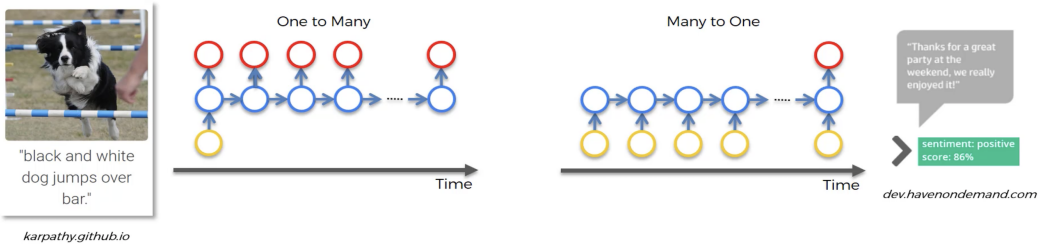


Figure 4: RNN one to many and many to one relationship examples

Figure 5 and 6 demonstrate examples of RNN models used with *many-to-many* relationships. In Figure 5 the sentence with multiple words in English language is being translated to Armenian sentence with many words. What is interesting to see is that while there is only single word difference between the two sentences in English, (boy versus girl) the corresponding translated sentences have more than one word difference. The two outputs differ entirely from each other and this is a result RNN many-to-many implementation. To translate these sentences properly the least what the model needs to know is the gender behind the words boy and girl and then adjust the rest of the translation appropriately. This is what RNN is doing with using short-term memory and getting context about the inputs that helps to predict the output.

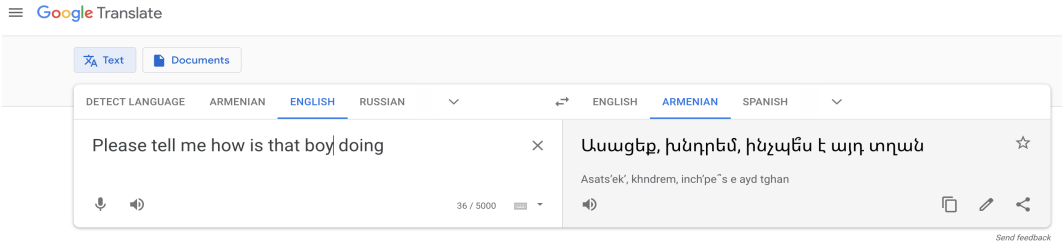


Figure 5: RNN one-to-many and many-to-one relationship examples

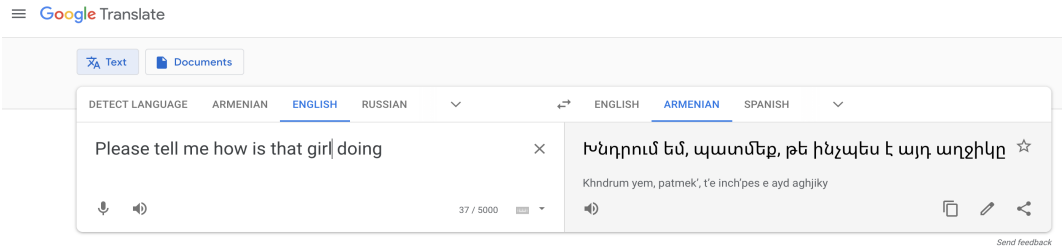
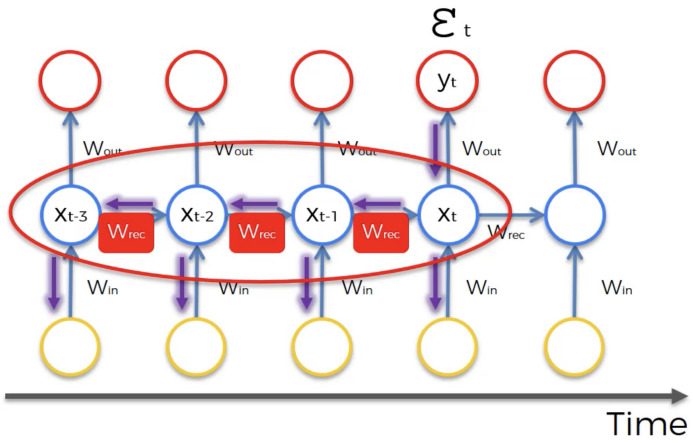


Figure 6: RNN one to many and many to one relationship example

3.1 The Vanishing Gradient Problem

Like in any other Neural Network in case of RNN there is process that involves an optimization technique, for instance a Gradient Decent algorithm, that helps to determine the optimal weights for each epoch that minimizes the cost function. On one hand the information flows from the input layers to the output layer and on the other hand, the error is calculated that is back propagated through the network to update the weights. Unlike ANN and CNN, in case of RNN hidden layers not only give an output but also



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \tag{3}$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \tag{4}$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \tag{5}$$

Figure 7: RNN Optimization Process and Vanishing Gradient

feed themselves. So, Neurons are also connecting through each other thanks to a short term memory which can be seen in Figure 7 which is a simplified visual representation of RNN process where we have many-to-many relationship. ε_t is one of the many output's cost function values that form a *time series* of $\{\varepsilon_1, \varepsilon_2, ..., \varepsilon_{t-1}, \varepsilon_t, \varepsilon_{t+1}\}$. Let us focus at one specific cost function, ε_t, corresponding to the time period t and the goal is to propagate this error back to the network to update the weights of all neurons that have participated in the process of calculated this error value. So, not only directly but also indirectly contributed, to the error, neurons need to be updated which depends also at the number of time periods one considers to go back in the past. In this figure, the number of considered 3 lagged (t-3, t-2, t-1) periods neurons are contributing to the calculation process. In the right part of Figure 5 we can see the optimization equations used to calculate the weights and the $\mathbf{W}_{rec}$ stands for the **Weight Recurrent** used to connect the hidden layers to themselves $(X_{t-3}, X_{t-2}, X_{t-1})$ in a temporal loop that calls under the circled area. Given that in this process each of this values are multiplied with the same weight, W_{rec}. In the very initial epoch, the weights are randomly chosen values and in case this chosen values are very small, multiplying them with W_{rec} many times, the

gradient becomes less and less and at some point the gradient will vanish because the lower the gradient is, the harder is to update weight which means the slower will be the process. Moreover, there is a domino effect and one improperly calculated weight effects the calculation of the remaining weights and makes them inaccurate as well, given that they all are related. Additionally, the other extreme is possible as well, that is having *exploding gradient* when the W_{rec} is large. To sum up, if the W_{rec} is small ($W_{rec} < 1$) then there is a big risk of **Vanishing Gradient Problem** whereas if the W_{rec} is large ($W_{rec} > 1$) then there is big risk of **Exploding Gradient Problem**. There are different ways to solve these two problems. Vanishing Gradient Problem can be solved for instance with *Weight Initialization*, *GRU* or a very popular technique called *Long Short-term Memory Networks (LSTMs)* whereas Exploding Gradient Problem can be solved with for instance *Penalties*, *Weight Initialization* and so forth.

3.2 RNN process

What RNN does is that it translates the provided inputs to a machine readable vectors. For example, if we aim to use RNN that will classify a review consisting of multiple words as negative or positive then the RNN will translate each word to a vector. Then the system processes each of this sequence of vectors one by one, moving from very first vector to the next one in a sequential order. While processing, the system passes the information through the hidden state (memory) to the next step of the sequence. This processes is visualized in Figure 8.

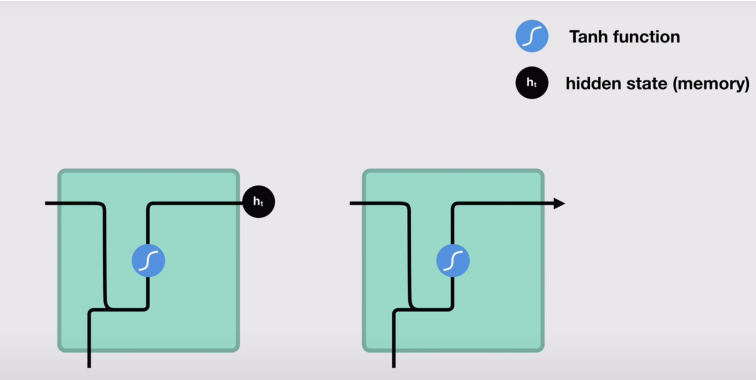


Figure 8: RNN Information Flow

Once the hidden state (h_t) has collected all the existing information in the system until time period t, it is ready to move towards the next step and in this newer step the hidden step is classified as *previous hidden state* defined by the letter (h_{t-1}). What happens is that this hidden state that caries the memory from the previous step is then

combined with this step's input (X_t) to form a new vector, as shown in Figure 9 [1] which contains information about the current vector and the previous inputs. Once, this joint vector passes through the Activation function (Hyperbolic Tangent Activation Function or Tanh function), a new hidden state (h_t) is created that can be passed to the next step. In this process, the Tanh function is used because it is one of the activation functions that transforms vectors in such a way that all the elements of this vector fall in the range of [-1,1]. In this way we avoid having too large or too small values that are transmitted through the system and cause problems.

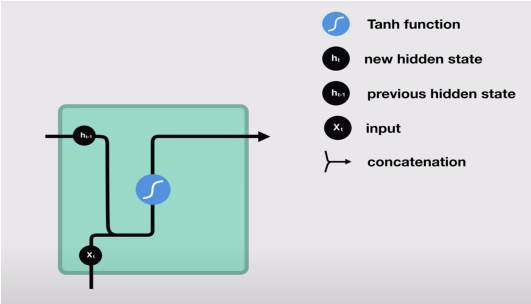


Figure 9: RNN Information Transition

3.3 Long Short-Term Memory (LSTM)

As it was mentioned earlier, when the recurrent weight is small, this can cause Vanishing Gradient problem which means that the weights in the far left in figure 7 are updated slower than the weights in the far right part which means that some of the right hand-side neurons will have relatively more influence on the output than the ones in the far left. Hence, the entire training of the network will be inaccurate and some part of an important information will be lost in the process and will not be kept in the short-term memory.

LSTMs have the same information flow as usual RNNs with the sequential information transition where data propagates forward through sequential steps. The difference between the usual RNN and LSTM is the set of operations that are performed on the passed information and the input value in that specific step. This process is visualized in Figure 10. These set of various operations gives the LSTM the opportunity to keep or forget parts of the information that flows into that particular step. The information in LSTMs flows through its *gates* which are LSTMs' main concepts; *forget gate*, *input gate*, *cell state*, and *output gate*.

[1] https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

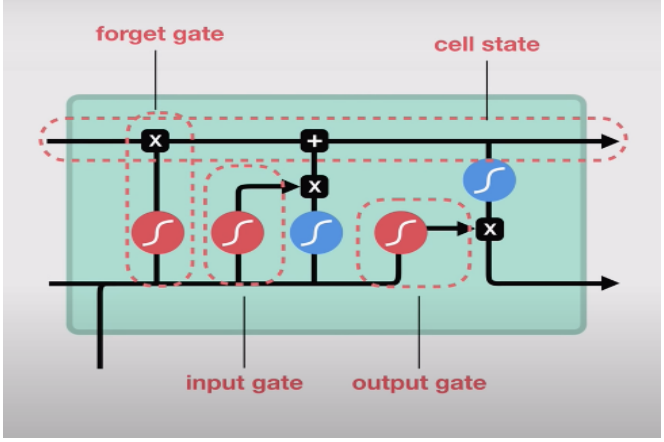


Figure 10: LSTM Information Transition

3.3.1 Forget Gate

The process starts with the transition of the information that comes from the previous step. This gate decides what information should be thrown away or kept. Information from the previous hidden state and information from the current input is passed through the Sigmoid function. Values come out between 0 and 1. The closer to 0 means to forget, and the closer to 1 means to keep. The Sigmoid function transforms vectors in such a way that all the elements of this vector fall in the range of [0,1] which helps to regulate the network by updating or forgetting data because any number getting multiplied by 0 is 0, causing values to disappear or to be "forgotten" and any number multiplied by 1 is the same value therefore that value stays the same or is "kept". In this way, the network can learn which data is not important therefore can be forgotten or which data is important to keep. This part of the process is described by the first column in the left part of Figure 10.

3.3.2 Input Gate

In this step, when the unnecessary info is forgotten and the necessary info is kept, the candidate state is determined that goes to through the *cell state*. This part of the process is described by the second and third columns in the left part of Figure 10. First, we pass the previous hidden state and current input into a Sigmoid function. That decides which values will be updated by transforming the values to be between 0 and 1. 0 means not important, and 1 means important. You also pass the hidden state and current input into the Hyperbolic Tangent function to squish values between -1 and 1 to help regulate the network. Then you multiply the Hyperbolic Tangent output with the Sigmoid output. The Sigmoid output will decide which information is important to keep from the Hyperbolic Tangent Function output.

3.3.3 Cell State

Once all the sources of information are processed then the cell state can be determined. First, the cell state gets pointwise multiplied by the forget vector. This has a possibility of dropping values in the cell state if it gets multiplied by values near 0. Then we take the output from the input gate and do a pointwise addition which updates the cell state to new values that the neural network finds relevant. That gives us our new cell state.

3.3.4 Output Gate

The output gate decides what the next hidden state should be. Note that the hidden state contains information on previous inputs and is also used for predictions. First, we pass the previous hidden state and the current input into a Sigmoid function. Then we pass the newly modified cell state to the Hyperbolic Tangent function. We multiply the Hyperbolic Tangent output with the Sigmoid output to decide what information the hidden state should carry. The output is the hidden state. The new cell state and the new hidden is then carried over to the next time step.hidden state and cell state are transmitted to the next time step.

3.4 Gate Regulating GRU

GRU is another method that solves Vanishing Gradient Problem for RNNs. The GRUs are the newer generation of RNNs and very similar to LSTMs. One of the biggest differences between the GRUs and LSTMs is that GRUs have no cell state and use the hidden state to transfer information. They have only two gates, a *reset gate* and *update gate*.

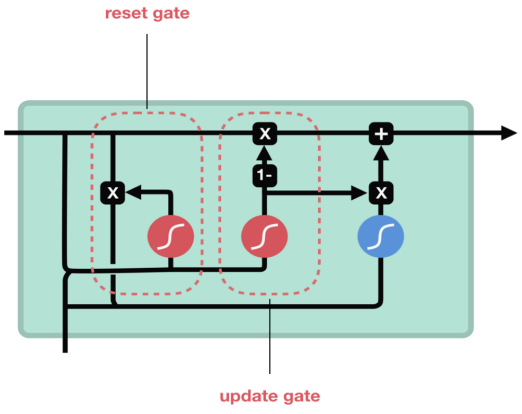


Figure 11: GRU Information Transition

4 Results and Evaluation

For the entire analysis, from data preparation till evaluation of the results, we will use *Python* and we use Deep Learning Python libraries *Tensorflow* and *Keras* to train and validate the LSTM model. More specifically, from the Keras library, we will use *Sequential* module that will allow us to create a Neural Network object with sequential layers. Moreover, we will use the *Dense* module to add an output layer and the *LSTM* class to add LSTM layers. Finally, we will use *Dropout* modul from Keras to add dropout regularization to the model. We start the process by adding an extra dimension to the training data given that we would like to use additional indicators next to the past stock prices to predict the future stock prices.
In this project, we aim to predict the stock prices which is a continuous output value, hence we have a regression rather than classification problem. So, now we initialize this regressor as an object with sequential layers. Then we add a LSTM layer to this earlier initialized RNN model. Given that the stock price prediction is a quite complex task, we like to have a model with high dimensionality that can capture upward and downward trends in the stock price, we use large number of, 50, LSTM units or neurons per LSTM layer. We then add a Dropout for regularization to ignore part of the neurons in the LSTM layers. More specifically, we drop the 20% of neurons in the LSTM layer during the training process. The model has 5 LSTM layers, which means that after adding the first LSTM layer, we will add another 3 layers to it.

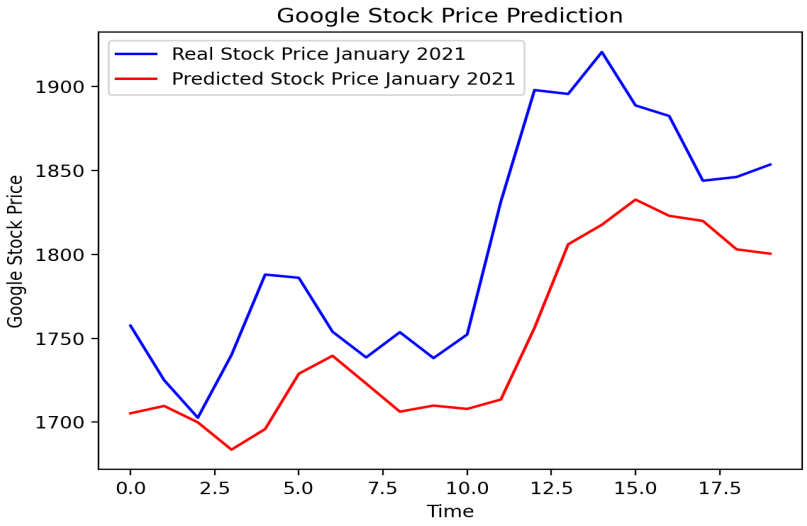


Figure 12: Predicted versus Real Stock Prices

As it was mentioned in before, our goal in this project is to predict the stock prices, so we have a continuous output variable, hence we have a regression problem. Therefore, we

will use the *Mean Squared Error* as a loss function, to evaluate the model. Additionally, we will use the *Adam Optimizer* as loss function optimizer. To have converging model, we will have 100 epochs to train the built regressor model with X_{train} and y_{train}. Finally, we will use a batch size of 32 such that the the weights will be updated per every 32 stock prise.

Figure 12 compares the real stock values and predicted stock values that are generated using RNN model, for the test period which is the first month of 2021. We observe that the RNN based on LSTMs was able to properly predict all upward and downward trends because we see that the red line corresponding to the predicted stock prices follows the same pattern as the blue line which corresponds the real stock prices.

References

[1] Jay Kuo C.-C.(2016). Understanding Convolutional Neural Networks with A Mathematical Model 1–21.

[2] Glorot, X., Bordes, A., Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks *International Conference on Artificial Intelligence and Statistics, 15(15)*

[3] Rogerson, R. J. (2015). Adam: A Method For Stochastic Optimization. *3rd International Conference on Learning Representations (ICLR2015), 36(1)*, 1–13.

[4] LeCun, Y., Bengio, Y., Hinton, G., (2015). Deep learning. *Nature, 521*, 436 – 444.

[5] Sherstinsky, A., (2018). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Termemory (LSTM) Network, *CoRR*, 1 – 43.