# Predicting the Nikkei 225 Index Price, using Long Short-Term Memory with TensorFlow and Keras in Python

School of Engineering
Zurich University of Applied Sciences

Pascal Aigner, Maurice Gerber, David Jung and Basil Rohr

June 2021

**Abstract** This short project work gives a simplified overview of using Long-Short Term Memory to predict a stock price. The code is written entirely in Python and uses TensorFlow and Keras. The focus is on the input parameters for the algorithm and their effect on the final prediction. There is very little theory in this thesis. Therefore, a solid knowledge of stochastic processes, neural networks and statistics is assumed. At the end, the prediction made with LSTM is compared with a simple ARMA model. The entire code is published on:
`https://github.com/mauricegerber/neural_networks_econometrics.git`.

# 1 Introduction

There has been extensive research on predicting a stock price for a long time. It is an extremely interesting field, albeit one that involves very complex issues. However, it is important to note that the stock market is highly volatile and considered as unpredictable, no matter how much data is trained into a model, there are many other factors that cannot be controlled. Nevertheless, in this short paper, an approach for a possible accurate prediction using recurrent neural network and Long Short-Term Memory (LSTM) is presented. The focus is more on the practical implementation than on the theoretical and mathematical background. The code is entirely written in Python, and is based on already existing algorithms using TensorFlow and Keras. Keras is a high-level neural networks library that is running on the top of TensorFlow. Whereas TensorFlow is an end-to-end open source machine learning platform developed by the Google Brain Team and released in 2015. The aim of this work is to explain the code and its individual components in order to investigate the influence of its parameters on the final result. The end result is a predicted stock price at a certain point in the future. In order to continuously review the code and its results, historical data of the Nikkei 225 Index is used [1].

# 2 Neural network and Long Short-Term Memory

The goal of this chapter is to provide a brief overview of artificial neural networks (ANN), with a focus on neural networks used for prediction. In particular the Long Short-Term Memory (LSTM), which will be used in the code, is introduced.

Neural networks are a series of algorithms very similar to the human brain, which are used to recognize patterns. They interpret sensory data by machine perception, labelling or clustering raw data. In general, the ANN consists of three types of neurons: Input neuron, hidden neuron and output neuron. The input layer receives the input signals and passes them on. In a next step, the input value is passed through a series of hidden layers, which transforms the input value. Within each node, weightings are applied. In a node, for instance, an input value can first be multiplied by the weighting and then a constant may be added. The last layer of the neural network is called the output layer and represents the prediction of a desired variable. More precisely, the process of calculating the output comprises two steps. First, the input values of a neuron are multiplied by their weights and then summed up, eventually they get biased. Then the result is passed to the so-called activation function. This output can then be passed on and becomes the input for further neurons. When the multi-layered perceptron has many hidden layers, it is called a deep neural network depicted in figure 1.

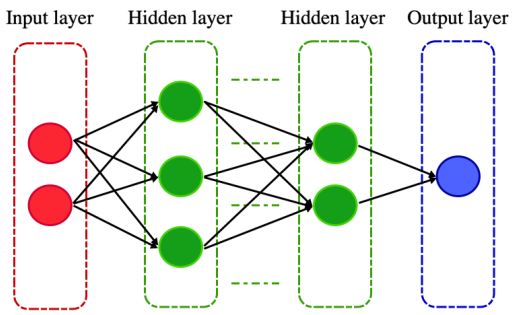


Figure 1: Illustration of a deep neural network

In principle, there are two main processes in the neural network that are frequently used, namely feed-forward and back-propagation. Feed-forward is an algorithm for calculating output values based on input values, while back-propagation is an algorithm for training neural networks based on errors derived from output values. LSTM is particularly suitable for the classification, processing and prediction of time series for time delays of unknown duration. It trains the model by applying back-propagation. There are three gates in an LSTM network:

Input Gate - Determines which value from the input should be used to modify the memory. So it is defined which values are transmitted and how these values are weighted.

Forget Gate - It is decided which details from the block are discarded. To do this, the previous state and the new input are considered and a decision is made whether the individual values are to be retained or discarded.

Output Gate - The input and memory of the block are used to determine the output. It is again decided which values are transferred and how they are weighted [2] [3].

# 3 Data

In this paper, daily historical data from the Nikkei Heikin Kabuka Index, short Nikkei 225 or just Nikkei, is used. The data stems from the period 01.01.1995 to 01.01.2021. The Nikkei is Japan's leading index and the most important stock index in Asia. The Nikkei measures the performance of 225 large, publicly owned companies in Japan traded on the Tokyo Stock Exchange (TSE). The reason for choosing this data is because the index price does not show a clear linear trend. Most stock or index prices, such as the SP500, show a clear positive upward trend since their introduction. The Nikkei, on the other hand, is rather volatile in comparison and does not have a clear tendency within the used time period as seen in figure 2. This has a positive effect on the training of the data later on, it will make the learning more robust as well as give you a chance to test how good the predictions are for a variety of situations [4].

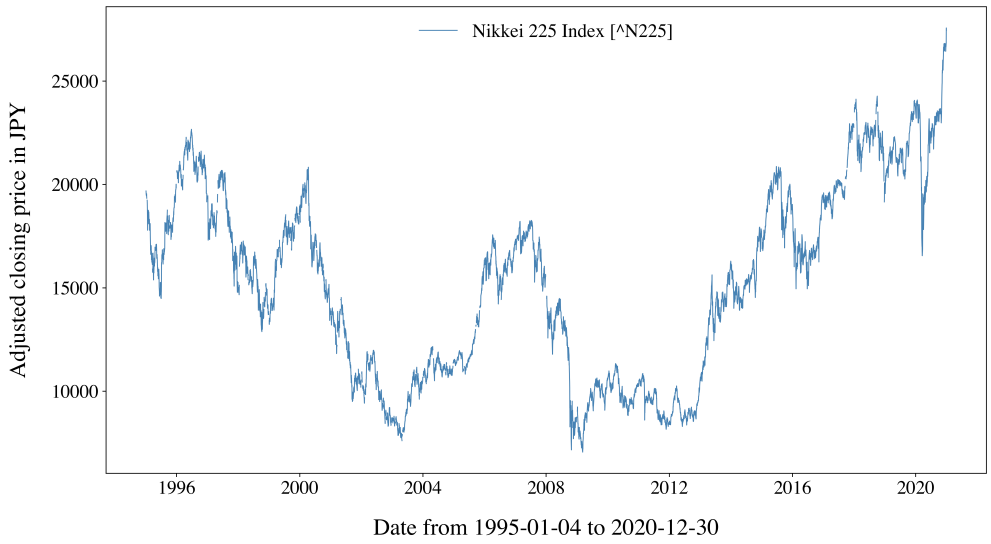


Figure 2: Nikkei index price

# 4 Structure of the algorithm

In this chapter the structure of the code will be explained step by step. The chapter is divided in several sub-chapters beginning with the input. The input shows how the data should be prepared and what's important to note. Afterwards the different parameters and their impact will be explained. In the end the different optimizations were analyzed and their results compared.

## 4.1 Input

The code can be used for any stock, index, fund, cryptocurrency, etc. as long as historical data is available. In this case the historical index price of the Nikkei 225 is used as explained before in chapter 3. To train the LSTM model the following values are needed; 'adjclose', 'volume', 'open', 'high' and 'low'. As well it's important to make sure that the data is sorted by date, because the order of the data is crucial in time series modelling. To get the historical data directly instead of uploading a CSV file there are various possible approaches, such as Quandl, Ritoku, Alpha Vantage, Yahoo Finance and so on. In this work the data has been imported direct via a free API from Yahoo Finance, with the `yahoo_fin` package, displayed in listing 1.

```
from yahoo_fin import stock_info as si
si.get_data('^N225', '01.01.1995', '01.01.2021')
```

Listing 1: Import historical data from Yahoo Finance

After importing the dataset, the dataset is split into a training dataset, which is used to train the model, and a test dataset to evaluate its performance. The split is done in a ratio of 80% to 20%, with the training data containing 80% of the total data set. This ratio is commonly used when working with financial time series. Another important step is scaling the input values. Unscaled input variables can result in a slow or unstable learning process, whereas unscaled target variables on regression problems can result in exploding gradients causing the learning process to fail. The code for scaling is shown in listing 2.

```
for column in feature_columns:
    scaler = preprocessing.MinMaxScaler()
    df[column] = scaler.fit_transform(np.expand_dims(df[column].values, axis=1))
    column_scaler[column] = scaler
```

Listing 2: Scale the data to a value within 0 and 1

## 4.2 Parameters

Now that the theoretical background and the data preparation have been explained, a closer look is taken at the parameters and their effects on the prediction. For the sake of time, only the data from the past 10 years are used. This allows us to produce the output with several different parameters in less time than with data from 26 years. The training data set and test data set ratio stays 80% to 20%, which means the following displayed test data is from the past two years.

**lookup_step**: It is the future lookup step to predict, the default is set to 1 which would mean the next day. If it's set to 20, it means the price in 20 days will be predicted. In this chapter the `lookup_step` is set to 1.

**n_step**: An integer which indicates the historical sequence length which is used to train the model, also known as window size. A sequence data is needed to feed the neural network, set 50 for `n_step` for example means that 50 days of index prices are used to predict the next lookup time step. The value of `n_step` can have a significant impact on the final prediction. In figure 3 several values are shown for `n_step`.

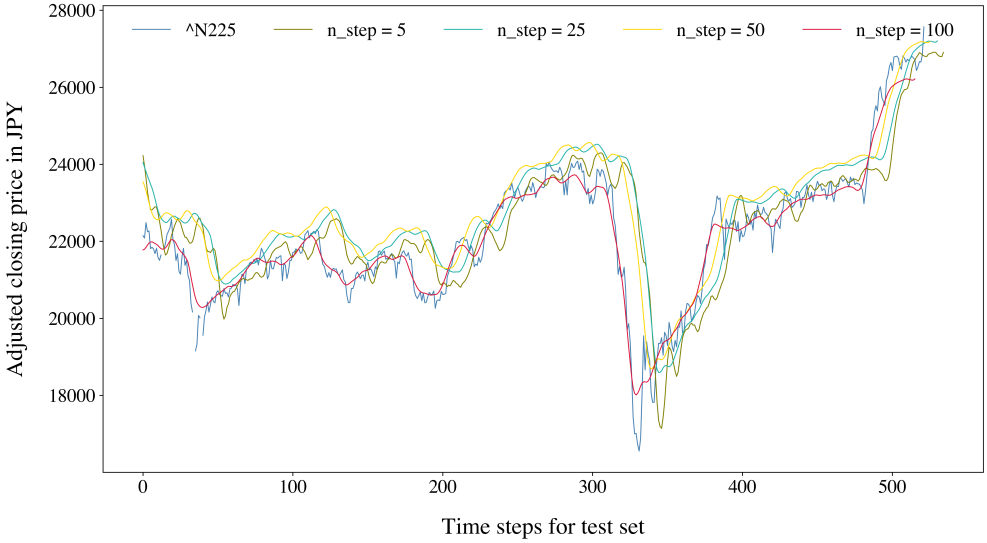


Figure 3: Different `n_step` values for the same model

Figure 3 shows that the best prediction is made with a `n_step` value of 100. It is clearly visible that with more training data a stronger model can be created, i.e. with 100 previous data points the next data point can be predicted more accurately than with only 5 data points. To further underscore the visual insight, the Sharpe ratio was calculated for each `n_step` value shown in table 1.

| **n_step = 5** | **n_step = 25** | **n_step = 50** | **n_step = 100** |
|---|---|---|---|
| 0.531 | 0.979 | 1.162 | 1.514 |

Table 1: Sharpe ratio for different `n_step` values

**dropout**: The dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting the training data. This means that for a given probability (dropout rate), the data on the input connection to each LSTM block will be excluded from node activation and weight updates.

**n_layer**: A higher number of layers results in more trainable parameters. However the Sharpe ratio decreases with more layers, as shown in table 2.

| **n_layer = 2** | **n_layer = 3** | **n_layer = 4** | **n_layer = 5** |
|---|---|---|---|
| 793,857 | 1,319,169 | 1,844,481 | 2,369,793 |
| 1.615 | 1.457 | 1.412 | 1.198 |

Table 2: Total parameters and Sharpe ratio for different `n_layer` values

**batch_size**: `batch_size` is how many data samples being considered in a single time step. The value for the `batch_size` has a significant impact on the time needed to train the model. The larger the value, the faster the calculations. However, the output does not show a large difference as seen in figure 4, neither is there a significant difference in the Sharpe ratio shown in table 3.

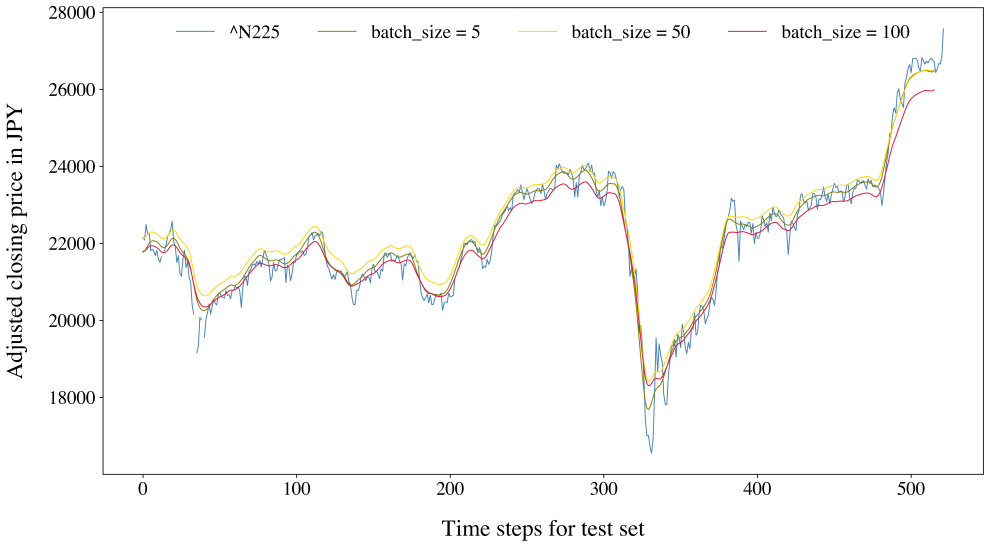


Figure 4: Different `batch_size` values for the same model

| **batch_size = 5** | **batch_size = 50** | **batch_size = 100** |
|---|---|---|
| 1.474 | 1.546 | 1.569 |

Table 3: Sharpe ratio for different `batch_size` values

**loss**: The purpose of loss functions is to compute the quantity that a model should seek to minimize during training. There are several loss functions available in Keras. To select the best fitted loss function for the data, the daily returns in percentage are visualised in figure 5. There is a mean of 0.000158, as well as the upper $[3\sigma]$ and lower $[-3\sigma]$ lines, which refer to the triple standard deviation. There are some outliers visible and therefore the Huber loss function has been selected. The Huber loss function is a combination of the mean absolute error and mean square error, therefore it is less sensitive to outliers in data than the squared error loss, which means it's a more robust function.

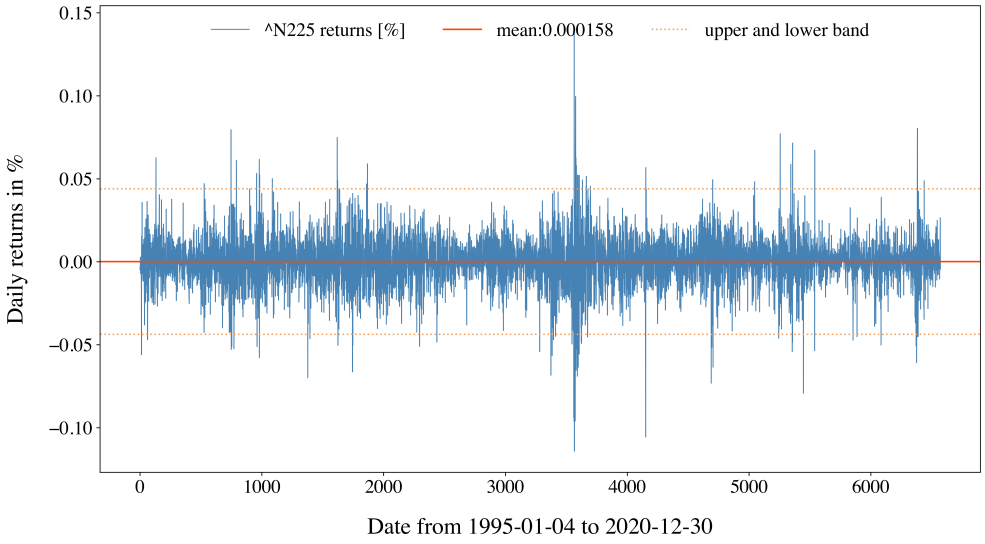


Figure 5: Daily returns of index price in %

### 4.3 Optimization

There are several optimizers for the Keras package. In this chapter, a rough overview of some possible optimizers used are given. For a more detailed explanation and the mathematical formulae, reference is made to the corresponding papers on the individual algorithms. In figure 6 it can be seen that, except for the SGD algorithm, all the others have a similar output. Nevertheless, there is a vertical shift depending on the iterations, so there is no clear result as to which optimizer gives the best prediction.

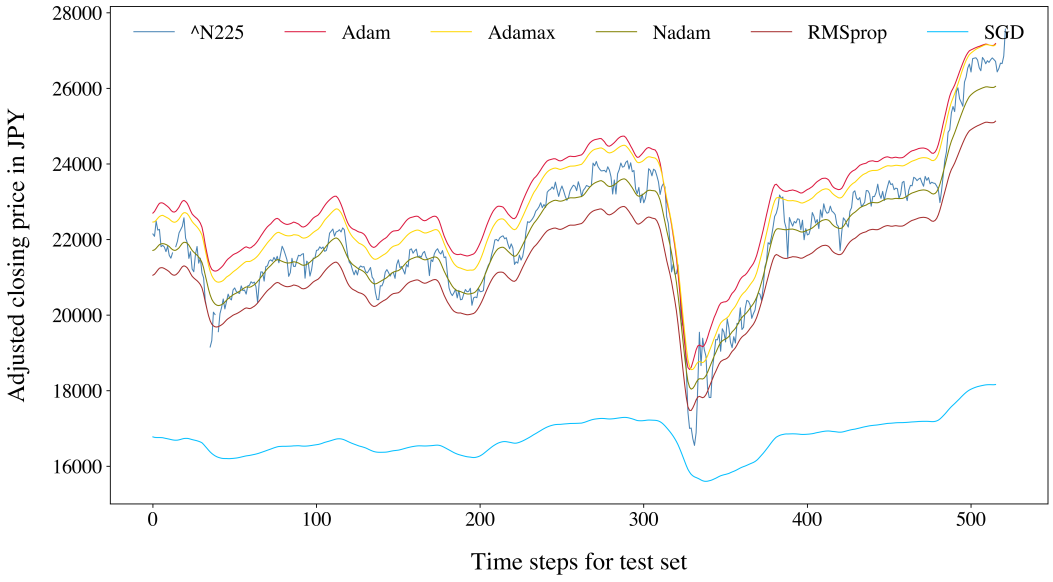


Figure 6: Different optimizations for the same model

**Adam**: The Adam optimizer is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. According to Diederik P. Kingma and Jimmy Lei Ba [5], the method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

**RMSprop**: It is an unpublished adaptive learning rate optimizer proposed by Geoff Hinton [6]. The motivation is that the magnitude of gradients can differ for different weights, and can change during learning, making it hard to choose a single global learning rate. RMSProp tackles this by keeping a moving average of the squared gradient and adjusting the weight updates by this magnitude.

**Nadam**: Much like Adam is essentially RMSprop with momentum, Nadam is Adam with Nesterov momentum. As investigated in the paper 'Incorporating Nesterov Momentum into Adam' by Timothy Dozat [7], the conclusion is that in most cases the improvement of Nadam over Adam is fairly dramatic.

**Adamax**: Similarly to Adam, a variable epsilon is added for numerical stability, especially to get rid of division by zero when the exponentially weighted infinity norm equals zero [5].

**SGD**: Stochastic Gradient Descent is an iterative optimization technique that uses minibatches of data to form an expectation of the gradient, rather than the full gradient using all available data. SGD reduces redundancy compared to batch gradient descent - which recomputes gradients for similar examples before each parameter update - so it is usually much faster [8].

## 5 Final model

With the previous discussion of the various input values, the optimal values can now be applied to create a final model. The parameter values listed below will be the settings of the model.

```
# Data
ticker = "^N225"
start_date = "01.01.1995"
end_date = "01.01.2021"
# Days into the future (y)
lookup_step = 1
# Days back (X), window size or the sequence length
n_steps = 100
# Test size
test_size = 0.2
# Feature column
feature_columns=['adjclose', 'volume', 'open', 'high', 'low']
# Layers
n_layers = 2
# dropout
dropout = 0.3
# Optimizer
optimizer = "RMSprop"
# Loss
loss = "huber_loss"
# LSTM cell
cell = LSTM
# LSTM neurons
units = 256
# Batch size
batch_size = 100
# Epochs
epochs = 4
```

Listing 3: Input values for the final model

As previously described in chapter 4.3, the prediction may shift vertically due to the optimization. To overcome this problem, one could run the code a large number of times to get the approximate mean value close to the actual price. To demonstrate this behaviour, the model was created 21 times with the same setting as in figure 7 displayed. For the sake of time and computer power it is only done 21 times. It is clearly visible that the mean value is getting close to the actual price which means a very accurate prediction is made.

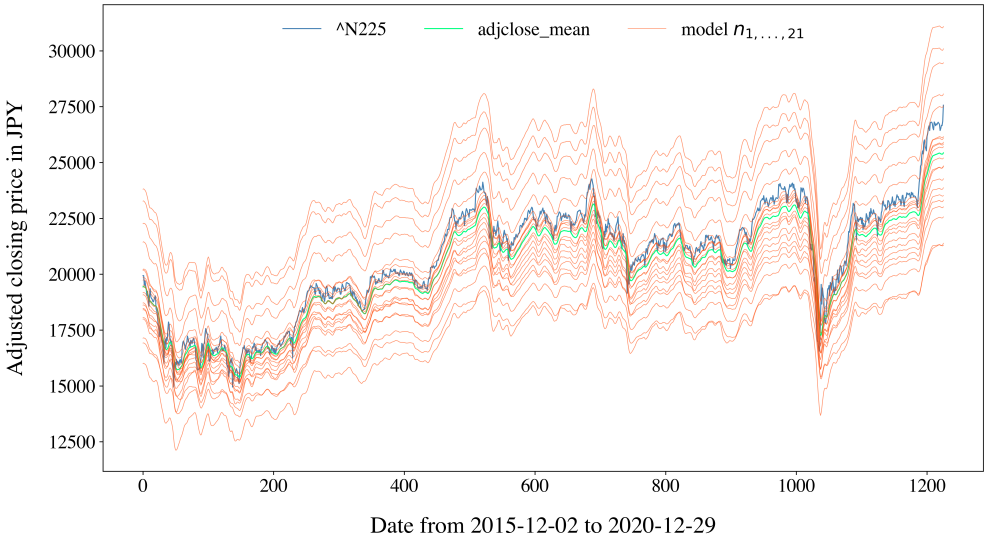


Figure 7: 21 iterations of the final model

The summary of the model is shown in table 4. It includes to total of the trained parameters [793,867] as well the different layers and it's number of trainable parameters of each layer.

| **Layer (type)** | **Param #** |
|---|---|
| lstm_1 (LSTM) | 268288 |
| lstm_2 (LSTM) | 525312 |
| dense (Dense) | 257 |

Table 4: Model summary with a total of 793,857 trained parameters

# 6 Prediction

Having examined the various input variables to create a final model for the most accurate prediction, this model can now be applied to predict the index price in a given day in the future. As shown in listing 4 the saved model is now loaded in a new Python file. After scaling the 'adjclose' values, the predict function is applied. Afterwards the predicted index price is rescaled to get a value in it's original form (JPY).

```
# load the final model
newmodel = tf.keras.models.load_model('prediction.h5')

# scale the 'adjclose' values
column_scaler = {}
for column in df.columns.values:
    scaler = preprocessing.MinMaxScaler()
    df[column] = scaler.fit_transform(np.expand_dims(df[column].values, axis=1))
    column_scaler[column] = scaler
df = np.array([df])

# predict function applied and rescaling the value
y_pred = newmodel.predict(df)
y_pred = np.squeeze(column_scaler["adjclose"].inverse_transform(y_pred))
y_pred = y_pred.astype(int)
```

Listing 4: Predict the scaled values from the final model 'prediction.h5'

The following input values were chosen for the final prediction, as seen in listing 5. This means that the model is given about 20 years of historical daily data to train, it goes back 200 steps to predict the price 30 steps in the future. Therefore, when the trained model is applied to daily index data, a prediction for the next 30 business days is possible. The process of training the model and predicting the index price was replicated 10 times. The output can be seen in figure 8.

```
start_date = "01.01.1995"
end_date = "05.01.2021"
# Days into the future (y)
lookup_step = 30
# Days back (X), window size or the sequence length
n_steps = 200
```

Listing 5: Input values for final prediction

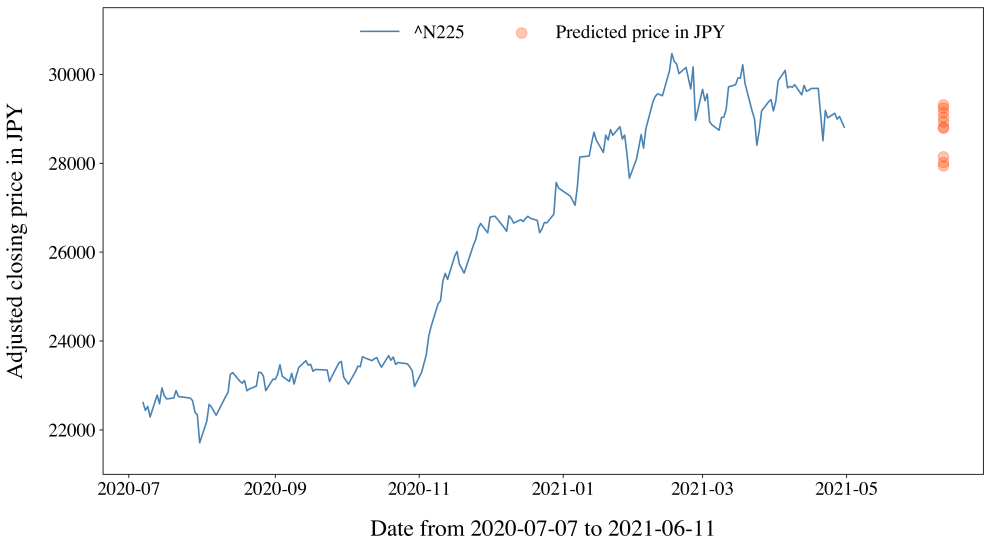


Figure 8: Predicted index price 30 business days ahead

## 6.1 ARMA vs. LSTM

After creating the rather complex LSTM model, the question came up whether the forecast could also be created with a simpler model such as an Auto Regressive-Moving Average (ARMA) model with a similarly effective result. Therefore, an ACF and PACF diagram must first be created to determine which parameters should be chosen for an possible ARMA model.

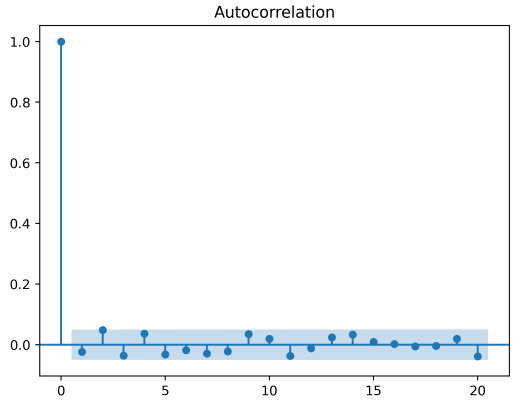


Figure 9: ACF plot

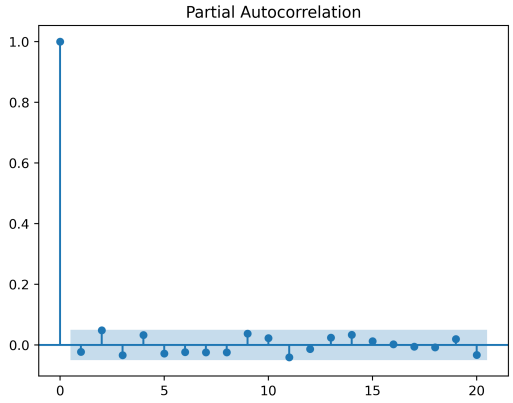


Figure 10: PACF plot

The ACF (figure 9) and PACF (figure 10) plot were generated in python with help of `statsmodels` library. The blue area pictures the confidence level which is set to 95%. In other words, if the point is outside the blue area you may say that with 95% probability is has a certain impact on values. If the bar is inside the blue area you may ignore this particular lag as most likely it is not relevant. The first point with index 0 has a height of 1. In fact the first lag is always equals 1, as it fully explains the current value.

```
1 model = ARMA(diff_data, order = (2,2))
2 model_fit = model.fit()
```

Listing 6: Create an ARMA model

As seen in the ACF (figure 9) and PACF (figure 10) plots, an ARMA(2,2) model seems to be the best option. After creating the model, the `model_fit.summary()` is shown in table 5. Since the p-values are less than the significance level of 0.05, it can be concluded that the coefficient is statistically significant.

| **param** | **coef** | **std err** | $P > \|z\|$ |
|---|---|---|---|
| const | 7.376 | 6.556 | 0.261 |
| ar.L1 | -1.303 | 0.301 | 0.000 |
| ar.L2 | -0.585 | 0.277 | 0.035 |
| ma.L1 | 1.286 | 0.295 | 0.000 |
| ma.L2 | 0.605 | 0.265 | 0.023 |

Table 5: ARMA model summary with coefficient, standard error and p-value

With the final ARMA model, a prediction for the next 30 business days can be made and compared with the LSTM forecast explained in chapter 6. Figure 11 displays both predictions and it can be clearly seen that the ARMA prediction matches the LSTM one.

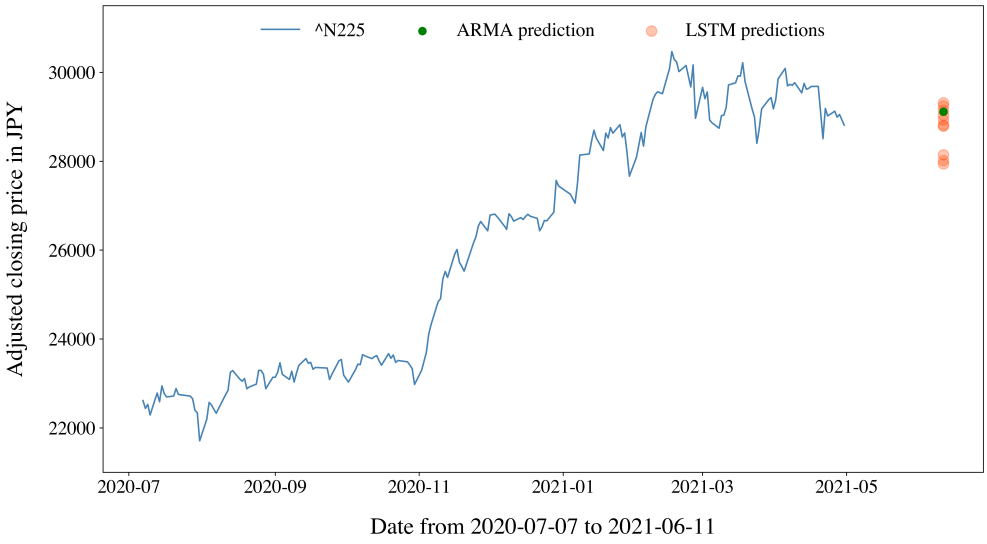


Figure 11: ARMA prediction compared with LSTM prediction for 30 business days ahead

# 7 Conclusion

Neural networks are a very interesting topic and there are just as many other uses for a neural network or LSTM. Nevertheless, using it to predict stock prices is perhaps not the most promising way to use it. Even though the work did not go very deeply into the mathematical background, it could easily be seen that simpler models, such as an ARMA model, also achieve a similar result. The effort to create the code and examine the different parameters and their effects was quite high compared to the result. More detailed work and a more mathematical approach could certainly lead to a more satisfactory result. The different optimizers and their effects could also be investigated more deeply. It is interesting to see what new possibilities a neural network brings with it and where this technology will take us in the next few years.

## References


[1] Abdou Rockikz. How to predict stock prices in python using tensorflow 2 and keras, 2021.

[2] Sepp Hochreiter and Jürgen Schmidhuber. *Long Short-Term Memory*. Technische Universität München, 80290 Munich, Germany, 1997.

[3] Christopher Olah. Understanding lstm networks, 2015.

[4] Yahoo! Nikkei 225 (n225) historical data, 2021.

[5] Diederik P. Kingma and Jimmy Lei Ba, editors. *Adam: A Method for Stochastic Optimization*, Published as a conference paper at ICLR 2015, San Diego, USA, 2015.

[6] Geoffrey Hinton, editor. *Neural Networks for Machine Learning*, 2012.

[7] Timothy Dozat, editor. *Incorporating Nesterov Momentum into Adam*, 2015.

[8] George Dahl Ilya Sutskever, James Martens and Geoffrey Hinton, editors. *On the importance of initialization and momentum in deep learning*, Proceedings of the 30th International Conference on Machine Learning, Atlanta, Georgia, USA, 2013.


## Listings



## List of Figures



## List of Tables