FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor’s thesis

Title:	Neural Networks for Modeling and Prediction of Stock Price Development
Student:	Martin Šír
Supervisor:	Ing. Mgr. Ladislava Smítková Janků, Ph.D.
Study program:	Informatics
Branch / specialization:	Knowledge Engineering
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of this work is to design a model based on neural networks for modeling the development of stock prices.

1. Get acquainted with the issues of technical analysis of stock prices. As part of the research, map both classical methods of technical analysis and methods using artificial intelligence techniques with emphasis on work using different types of artificial neural networks.
2. In collaboration with the supervisor, select one or more stocks to focus on. Create a dataset for the selected title/titles.
3. Select methods for model creation in the field of artificial neural networks.
4. Implement selected methods using existing tools.
5. Design and perform experiments for model verification, process their outputs and evaluate them.

Electronically approved by Ing. Karel Klouda, Ph.D. on 13 December 2021 in Prague.

Bachelor's thesis

NEURAL NETWORKS FOR MODELING AND PREDICTION OF STOCK PRICE DEVELOPMENT

Martin Šír

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Mgr. Ladislava Smítková Janků, Ph.D.
May 10, 2022

Contents

List of Figures

List of Tables

I would like to thank my supervisor, Ing. Mgr. Ladislava Smítková Janků, Ph.D., who assisted me with the management and organization of the thesis. I would also like to thank my family and close friends who supported, encouraged and motivated me.

Declaration

Abstrakt

Akciové trhy jsou náchylné na různé vlivy a proto je jejich predikce velmi náročná. Existuje mnoho přístupů, jak předpovídat vývoj cen akcií, nicméně žádný není univerzální. Tato práce se zabývá přístupem zvaným machine learning. Do totoho přístupu se dají zařadit modely CNN, LSTM, GRU, jejich obousměrné varianty a následné kombinace modelů. Dále jsou jednotlivé modely použity pro jednodenní a vícedenní predikci. V práci je upozorněno na nepoužitelnost jednodenní predikce, která je opakovaně použita za účelem predikce několika dní dopředu. Pro predikci byla vybrána data SXR8.DE, které kopírují index S&P 500. Následně jsou modely na těchto datech vyhodnoceny a na základě jejich přesnosti je vybrán model, který má největší úspěšnost za cílem maximalizovat profit z obchodování s akciemi.

Klíčová slova LSTM, GRU, CNN, CNN-LSTM, Bidirectional architectures, Stacked architectures, Kodér-Dekodér LSTM, Vícekroková predikce akcií

Abstract

Stock markets are vulnerable to various factors, and therefore their prediction is very challenging. There are many approaches to forecasting stock prices, but none is universal. This thesis focuses on an approach called machine learning. CNN, LSTM, GRU models, their bidirectional variants and model combinations can be included in this approach. Furthermore, the individual models are used for one-day and multi-day forecasting. The thesis points out the unusability of the one-day prediction, which is repeatedly used to predict several days ahead. SXR8.DE data, which replicates the S&P 500 index, was selected for the forecast. The models are then evaluated on this data, and based on their accuracy, the model with the highest success rate is selected in order to maximize the profit from stock trading.

Keywords LSTM, GRU, CNN, CNN-LSTM, Bidirectional architectures, Stacked architectures, Encoder-Decoder LSTM, Multi-step stock forecasting

List of abbreviations

ANN	Artificial Neural Network
AR	Autoregressive Model
ARCH	Autoregressive Cconditional Heteroskedasticity model
ARIMA	Autoregressive Integrated Moving Average
BPTT	Backpropagation Through Time
BRNN	Bidirectional Recurrent Neural Network
BVAR	Bayesian Vector Autoregression
CEC	Constant Error Carousel
CPU	Central Processing Unit
CSV	Comma-separated Values
CV	Cross-validation
EMA	Exponential Moving Average
ETF	Exchange Traded Fund
FNN	Feed-forward Neural Network
GARCH	Generalized Autoregressive Conditional Heteroskedasticity
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
LSTM	Long Short-Term Memory
MSE	Mean Squared Error
NLP	Natural Language Processing
NOR	NOT OR
ReLU	Rectified Linear Unit
RMSE	Root Mean Squared Error
RMSD	Root Mean Squared Deviation
RNN	Recurrent Neural Network
SARIMA	Seasonal Autoregressive Integrated Moving Average
SMA	Simple Moving Average
SVC	Support Vector Classifier
SVR	Support Vector Regressor
SVM	Support Vector Machine
tanh	Hyperbolic Tangent
XOR	Exclusive OR

Chapter 1

Introduction

Recently, stocks and financial markets have become the focus of interest for a large part of the population. Some people see investing as an opportunity to fight inflation, which has increased drastically in recent months. Others are not so focused on the immediate situation but rather on the long-term perspective. They try to make money in the long run, thus securing themselves and their family financially.

Investing comes with many risks, so people try to find ways to minimize those risks. They try to understand how the market works, how it reacts to inevitable fluctuations in the economy, or how it responds to the world. Other factors influencing the stock market are political decisions, interest rates, trends, international relations, currency exchange rates and the environment. There are many of these factors, and since financial markets are prone to fluctuations of any kind, their prediction is very difficult and almost unrealistic. However, any kind of help which can at least slightly determine the development of stocks is welcome by any investor.

Financial market forecasting is nothing more than time series forecasting. A time series is a sequence of data that is collected over time. Thanks to this fact, the time series can be relatively easily visualized on a graph with time on the X-axis and the data's values on the Y-axis. The nature of a time series implies that the data are ordered. In the real world, there is a large amount of data that can be expressed as a time series. For example, when facing a problem, at what time to go to the store in order to not wait in a queue. Most of the time, people look on the internet to find out the average number of people in the store each hour, which is nothing but a time series.

There are many methods that can be used to predict time series. Some of them are better and some worse. These methods differ in their approaches. One part of them follows a more traditional statistical and analytical path, while the other tries to use state-of-the-art technologies such as machine learning.

This thesis will discuss machine learning models such as LSTM, GRU, CNN and their combinations for predictions of one day and multiple days. The main goal of this thesis is to create a model which will have the best prediction accuracy. Data SXR8.DE, which is an ETF that replicates the S&P 500 index, was selected and used to train and validate each model.

1.1 Goals

This bachelor thesis aims to test models from the field of machine learning that can be used for stock forecasting. Furthermore, its goal is to find a model with the best accuracy in order to maximize profit. To fulfil the main aim of the thesis, the following sub-goals must be met:

- research the methods used for a stock market forecasting,

- explain the basics of the models and describe how they work,
- implement selected models,
- tune the hyperparameters of each model,
- propose experiments,
- compare implemented models,
- measure the accuracy of individual models for both single-day and multi-day forecasts,
- point out the problem of repeatable use of 1-day prediction,
- evaluate the best model,
- test the best model on a larger dataset.

1.2 Approach

Firstly, I research standard models used for stock prediction. From them, I choose the most used ones, which I also implement. The selected models include LSTM, CNN, CNN-LSTM, GRU, Encoder-Decoder architecture, Stacked architecture and Bidirectional architecture. Then for each model, the optimal hyperparameters are found. For the hyperparameter tuning, the KerasTuner tool is chosen. Subsequently, I present experiments for the prediction of 1 day and 5 days. The experiments are divided based on the shape of the input sequence. Then, the models are evaluated based on the RMSE value for the prediction of the given stock market, and further, the best model with the highest accuracy is selected. Next, the best performing multi-step model is compared with the best performing model for 1-day prediction. Furthermore, I present the flaws of the 1-day forecast, and then I test the best multi-step model and the best one-step model on a larger dataset.

1.3 Thesis structure

In the second chapter, called “Related work”, an overview of models and methods used for stock forecasting is presented. In the next chapter, the theoretical background for selected models is provided. Also, I introduce the measure function used to evaluate each model. In the fourth chapter, the implementation details are described. The used programming language, libraries and the reasons for such a choice are also mentioned. The next chapter, called “Data”, explains why particular data was chosen and how the data has been preprocessed. The next chapter describes how the best hyperparameters for each model were selected and which tool was used for this problem. The chapter “Proposed models” describes models, their variants and chosen hyperparameters. The following chapter says how the experiments are divided, and then the experiments are proposed. Also, the results of each model are provided. The next chapter discusses proposed experiments and evaluates a model for real-world use. The chapter highlights the problem that occurs when the 1-day prediction is reused. The one-step forecast is then compared with a multi-step prediction. Furthermore, it mentions future work that will be done. The last chapter summarises the whole thesis.

Chapter 2
Related work

This chapter serves to explore existing solutions that address the issue of stock market prediction. It also outlines the models that will be subsequently described, explained and implemented.

Among the methods that choose the traditional statistical way can be included models based on the principle of Moving Average (SMA, EMA, ...). These methods use a moving window from which the average is usually calculated, which can be further enriched with weights so that the last days have more influence on the result. These models are used for short-term forecasting rather than long-term forecasting. Since moving averages are calculated from a historical perspective, they are not predictive in nature, which may result in a randomness of the result. Other models that are based on an analytical approach include the autoregressive integrated moving average model (ARIMA), generalized autoregressive conditional heteroskedasticity model (GARCH), vector autoregression model (VAR) and Bayesian vector autoregression model (BVAR).

Ariyo et al. [1] gave an example of the ARIMA model. The paper mentioned that the ARIMA model is the most common model used for financial forecasting. It also described how the model works and said that the ARIMA model is quite robust and effective in short-term forecasting. It even stated that the model may have a better success rate for short-term forecasting than most popular ANNs techniques.

Samal et al. [2] mention two other models, namely Prophet and seasonal autoregressive integrated moving average model (SARIMA). The main difference between SARIMA and ARIMA models is that SARIMA can consider the seasonality of the data. For example, the model can predict stock price movements during significant events such as special holidays or some events that are repeating. Another model that was introduced in the paper is Prophet, which is a model that was developed by Facebook and is used for time series predictions. The model attempts to capture the seasonality of the data, which can sometimes be helpful for prediction, but the season does not always affect the time series. Thus, it is not very universal for every time series. However, the model works reasonably well for those that follow seasonality. Another thing the model can capture is the trend.

In their work, Eapen et al. [3] mentioned the use of CNN in combination with Bidirectional LSTM. Another model mentioned in the paper is the support vector machine regressor (SVR). The paper was inspired by the idea of SVM for time series prediction presented in the article [4]. In this paper, Meesad et al. claim that SVM along with neural networks are among the machine learning methods that can handle stock market prediction. SVM can be divided into two parts depending on which problem it solves. Their division is support vector regressor (SVR) which is used for regression and the other is support vector classifier (SVC) for classification. The paper also mentions the so-called Windowing functions, which are used to preprocess the data. The data is further sent to the machine learning model, whose goal is to detect the pattern of the dataset. There are many of these windowing functions and some give better results than others. However, Eapen et al. [3] tried the SVR that was mentioned in [4], but it turns out that CNN combined with Bidirectional LSTM gives better results than support vector regressor.

Hoseinzade et al. [5] come up with a CNNpred model that can be divided into two variants. The first is 2D-CNNpred, which takes as input a 2D matrix of data that has individual time points in its rows and features of a given market in its columns. This variant considers only one market, making it different from the second one. The second variant takes a 3D data matrix as input, enriched by one dimension compared to the previous one by stacking multiple markets. Thus, the result is a matrix with 2D matrices of individual markets next to each other. By doing this, the model is able to capture multiple different markets. The models are then tested on various markets to obtain the best model in general.

Another way to improve these models is to add news and social media posts that reflect what is happening globally. In general, most people who invest are emotional, so they immediately panic and sell their shares if alarming news occurs. This fact also impacts the resulting price of a particular stock. Thus, in their work, Gupta et al. [6] try to incorporate the sentiment of Twitter posts into the model, which tells whether a given tweet is positive (price rises), negative (price falls) or neutral (price stays the same) for the development of stock. This sentiment is further combined with historical data, and the result is sent to the LSTM model. The TextBlob tool is used for message sentiment, but it takes each post word separately, which may not give adequate sentiment results. However, according to the authors, the model mentioned above gives better results than the one without sentiment.

Another article that tried to involve the news in a prediction is [7]. The authors used Twitter data, news headlines and google trends data, which were merged with the historical data. The VADER tool was used to determine the sentiment of individual news headlines and Twitter data, which again uses a word-by-word approach, so the resulting sentiment may not match reality.

Khan et al. [8] tried using the Stanford NLP approach for sentiment, which differs in that it does not use a word-by-word method but looks at the text as a whole. It also divides the sentiment into five groups: more negative, negative, neutral, positive and more positive.

It is not necessary to use only historical data to predict stocks. In their work, Selvin et al. [9] mention an approach called Fundamental Analysis, which aims to analyse the company as a whole. It focuses on its expenses, earnings and other data that may convince an investor to invest in the company. This approach is suitable for long-term investing and says how much the investor believes that the company will succeed. This approach is individual, as each investor has different aspects that are important to them and therefore has its own way of valuing the company itself. The authors also mention linear models such as ARIMA, ARMA, AR and non-linear models like ARCH and GARCH. At the end of the article, ARIMA and other models like CNN and RNN are compared. From the results, it can be seen that CNN and RNN models give better results compared to the ARIMA model.

2.1 Chapter summary

In this chapter, the basic approaches for stock prediction were analysed. Furthermore, fundamental models from the field of machine learning were introduced. Based on this research, models that could be successful for the problem were selected. Based on the research, two basic approaches for stock prediction were found. From these papers, it was concluded that the machine learning approach is better.

Chapter 3

Theoretical background

This chapter explains and describes the models and architectures used in the practical part. It clarifies how the models work and what their strengths and weaknesses are. The chapter also defines the metrics used to evaluate the models.

3.1 Introduction to neural networks

The artificial neural network (ANN) is inspired by biological systems. The idea and creation of the ANN came with the desire to mimic the functioning of the human brain. The entire architecture is based on the human nervous system. However, instead of human cells, there are artificial neurons. These neurons can be considered the main building blocks of the ANN. A neuron can be seen as a working unit that has several inputs and produces output. Since there is a need for greater computational power, the structure of a neural network contains several of these neurons. These neurons are interconnected and communicate with each other in some way. These connections are weighted by weights that determine how much a given connected neuron is affected by another connected neuron. ANNs are powerful models primarily because they can produce both linear and non-linear dependencies between input and output data due to the larger combination of neurons. Each neuron then has an internal potential calculated from the values that enter it [10, 11].

3.1.1 Perceptron

Firstly, I introduce the single-layer perceptron model, which is the simplest ANN consisting of a single neuron. The structure can be seen in figure 3.1. Given the individual inputs $x_1, ..., x_n$, weights of inputs $w_0, ..., w_n$ and the internal potential ξ. The resulting internal potential of the neuron is calculated as follows:

$$\xi = w_0 + \sum_{i=1}^{n} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}, \tag{3.1}$$

where $\boldsymbol{x} = (1, x_1, ..., x_n)^T$, $\boldsymbol{w} = (w_0, w_1, ..., w_n)^T$ and w_0 is called intercept.

To calculate the output of a neuron, the internal potential is sent to a function called the activation function, which I will discuss later. In the case of a perceptron, the activation function is called a step function:

$$f(\xi) = \begin{cases} 1 & \text{if } \xi \geq 0, \\ 0 & \text{if } \xi < 0. \end{cases} \tag{3.2}$$

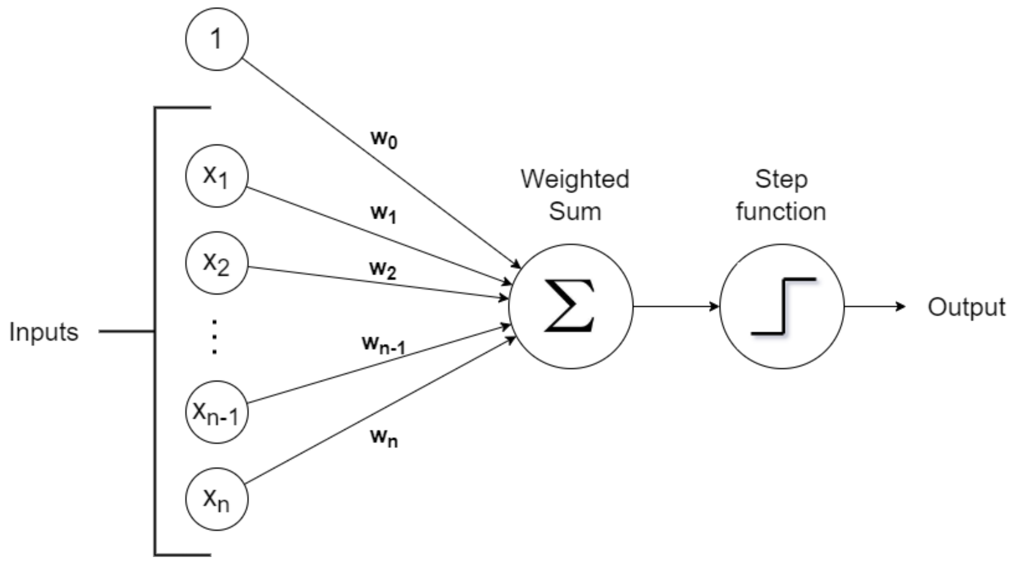


Figure 3.1 Perceptron model. Inspired by `https://www.cs.cas.cz/~petra/slides/2019_seninka_vidnerova/fig/single_perceptron.png`.

A limitation of the perceptron is that it cannot represent a function that is not linearly separable. Functions that a perceptron can represent are Boolean functions like AND, OR, NAND and NOR, but XOR is not a linearly separable function, so XOR cannot be represented by a perceptron [10, 11].

3.1.2 Activation functions

The activation functions are chosen to be nonlinear. They define the output of the neuron. There are many activation functions, and each one is suitable for a different kind of problem. One of the most famous is the sigmoid function. This function is suitable for solving a binary problem, for example. For classification into c classes, the so-called softmax function is used. For a regression problem, a linear function is used, or the ReLU function, which is similar to a linear function, but with the difference, that negative values are replaced by zero. The activation functions for neurons in hidden layers are mostly differentiable, which means that a derivative of them exists at each point. This fact is then used in neural network learning using backpropagation, which requires derivatives of activation functions. Backpropagation will be discussed in the next section [10].

3.1.2.1 Examples of activation functions

In this subsection the formulas of the basic activation functions are shown [12].
Sigmoid:

$$f(\xi) = \frac{1}{1+e^{-\xi}} = \frac{e^{\xi}}{1+e^{\xi}}. \tag{3.3}$$

Hyperbolic tangent:

$$f(\xi) = \tanh(\xi) = \frac{e^{\xi} - e^{-\xi}}{e^{\xi} + e^{-\xi}}. \tag{3.4}$$

Softmax:

$$f(\xi_i) = \frac{e^{\xi_i}}{e^{\xi_1} + ... + e^{\xi_c}}. \tag{3.5}$$

ReLU:

$$f(\xi) = \max(0, \xi). \tag{3.6}$$

3.1.3 Multi-layer perceptron (feed-forward ANN)

As already mentioned, the perceptron has some undesirable limitations that cause a small number of applications in practice. More neurons need to be added in order to get a network that can represent more complex and non-linear relationships. These neurons can be arranged in a single layer or multiple layers. If the final network contains multiple layers, it is called a multi-layer neural network. Thus, the general structure of a neural network includes an input layer, a varying number of hidden layers, and finally, an output layer. The neurons within a single layer are not interconnected.

On the contrary, the connected ones are those in different layers, specifically in adjacent layers. Thus, the output of a neuron of a particular layer is precisely determined by the inputs from the previous layer and the corresponding weights. This structure is called a feed-forward network (FNN) and can be seen in figure 3.2. Another type is a recurrent neural network (RNN), which I will explain later in the thesis [12].

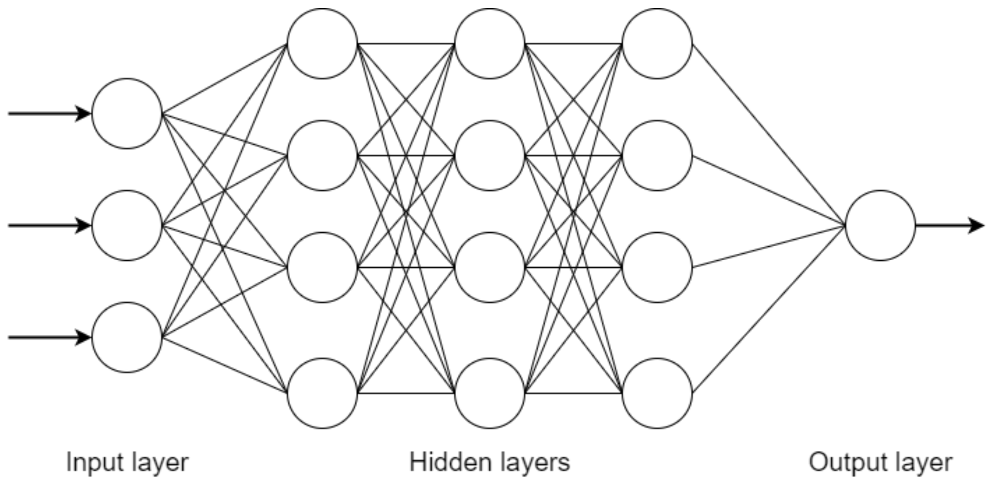


Figure 3.2 Configuration of a three-layer feed-forward neural network, each layer contains 4 neurons. Inspired by `https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/RNN-unrolled.png`.

3.1.4 Loss function

Another function I would like to mention is the loss function or sometimes called the error function. The loss function indicates how much the predictions differ from the actual values. So the function can evaluate the model's accuracy quite well and therefore describes how applicable the model is in the real world. The smaller the value returned by the loss function, the more accurate the model is. Thus, in practice, the goal is to minimize the loss function [12].

3.1.4.1 Examples of loss functions

There are many different loss functions that differ in what type of problem they are supposed to solve. The primary division of the issues is into two, namely regression and classification [12].

Firstly focus on the example of a regression problem. The most common loss function for regression problems is the squared error function. Given an outcome estimate $\hat{Y}$, an actual outcome Y, and a loss function L that has two parameters Y and $\hat{Y}$. For one prediction, the loss function is computed as:

$$L(Y, \hat{Y}) = (Y - \hat{Y})^2. \tag{3.7}$$

However, this error only measures the error for one predicted point. To calculate the error over all inputs, the average of these individual errors must be calculated. The resulting error function is called the Mean squared error (MSE) and is calculated as follows:

$$\frac{1}{N}\sum_{i=1}^{N} L(Y_i, \hat{Y}_i), \tag{3.8}$$

where N is the size of testing data.

Another variation is the root mean squared error (RMSE), also called the root mean squared deviation (RMSD). This loss function is similar to the previous one 3.8, but the difference is that the final result is the square root of the previous equation:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N} L(Y_i, \hat{Y}_i)}. \tag{3.9}$$

Another type of task is classification. For simplicity, I will only focus on binary classification. That is, one that divides the output into two classes. The most common loss function for binary classification is binary cross-entropy and is calculated as follows:

$$-Y \log \hat{p} - (1 - Y) \log (1 - \hat{p}), \tag{3.10}$$

where $\hat{p}$ is the estimation of the probability that the input vector belongs to the first class $(Y = 1)$.

There is also Categorical cross-entropy. However, it is not important for this thesis and you can read about it anywhere, so I will skip it. For further information, you can read the book [13].

3.1.5 Backpropagation

Backpropagation is an algorithm that allows a neural network to adjust weights and therefore learn. This style is called supervised learning because the required results are passed into the network, to which predictions should be as close as possible. These values represent the so-called teacher, i.e. learning with the teacher. Backpropagation is basically a gradient descent technique that allows the weights to be updated to optimize the average error of the model. The gradient determines the direction of the largest growth of the function, so to minimize the given loss function, the algorithm should go opposite to the gradient. In general, the whole algorithm works by passing input values to the network, which returns the prediction. Then the error is calculated and is sent from the output layer through the network back to the input layer. In each layer, each weight is updated according to the following formula:

$$w_x \leftarrow w_x - \alpha\Big(\frac{\partial J}{\partial w_x}\Big), \tag{3.11}$$

where w_x is the weight to be updated, α is the learning rate, and $\big(\frac{\partial J}{\partial w_x}\big)$ is the loss function gradient with respect to weight w_x.

The learning rate can be thought of as a number that indicates how large steps should the algorithm take when learning the network. It, therefore, expresses how fast the network learns. Of course, it is not possible to set a large learning rate with the expectation that the network will learn quickly, thereby reducing the time and computational requirements of the network. This is because if the algorithm takes large steps, it may reach and get stuck in a local minimum. Thus, it is common to choose a small number like 0.1 or 0.01. Often the alpha is chosen to be floating, so it gradually decreases with the number of iterations [10, 11, 12].

3.2 Recurrent neural networks

Another feed-forward network that I want to introduce is the recurrent neural network (RNN). The problem with standard feed-forward ANNs is that they are unable to capture historical data and remember it. Thus, previous inputs cannot affect the current inputs. This problem is called “The Problem of Long-Term Dependencies”. This is necessary, for example, when filling in words in the text. Sometimes, filling in is easy because the text is short. However, sometimes the network needs deeper context to know what word is most appropriate to fill in. RNNs were created to solve this problem. The idea was to create a model that would be able to have memory and therefore remember inputs from the past and thus influence the current output. RNNs have loops that allow them to persist the information and accordingly handle sequences of inputs that contain dependencies. These loops are sometimes called recurrent edges. The basic structure of RNNs is similar to that of ANNs. So it includes an input layer, hidden layers, and an output layer [14, 15].

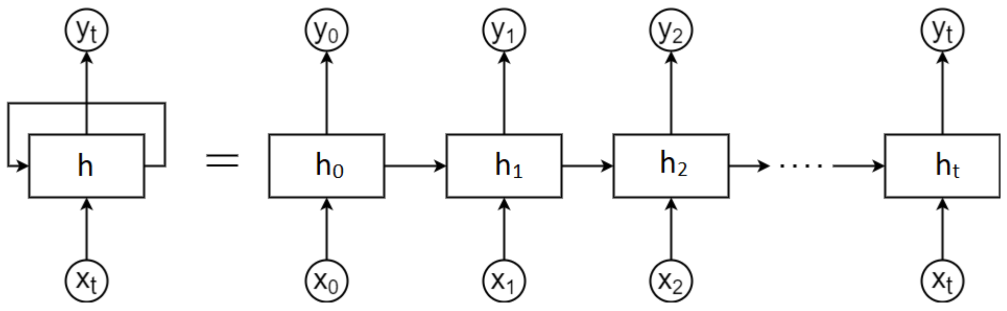


Figure 3.3 Unrolled recurrent neural network. Inspired by `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

In figure 3.3 you can see the basic structure of an RNN, where x_t is some input at time t, the output at time t is y_t and the hidden node values at time t is h_t. As you see, h contains a loop, which is used to pass information from the current step to the next one. Thus, the output y_t at each time t is not only determined by the current input x_t, but also by the hidden node values h_{t-1}. Current hidden state h_t at time t can be calculated as follows:

$$h_t = f(h_{t-1}, x_t). \tag{3.12}$$

The output y_t at time t is further computed from the hidden node values h_t at time t using a squashing function g.

$$y_t = g(h_t). \tag{3.13}$$

In practice, usually sigmoid or softmax is used, but it always depends on the type of task. The given structure looks hard to understand, as there is no simple one-way flow of information. However, it can be visualized by unfolding and thus understood as several connected copies of the same ANN [13, 14, 15, 16].

3.2.1 Backpropagation through time

Backpropagation through time (BPTT) is an algorithm used to learn recurrent neural networks. It was introduced in 1990 by Werbos et al. [17]. The algorithm treats the RNN in the same way as a feed-forward neural network. To treat the RNN as a feed-forward neural network, the RNN needs to be decomposed in time. Each layer represents a one time step, and all layers have the exact same weights, which is different from FNN. The resulting unrolled network behaves as an

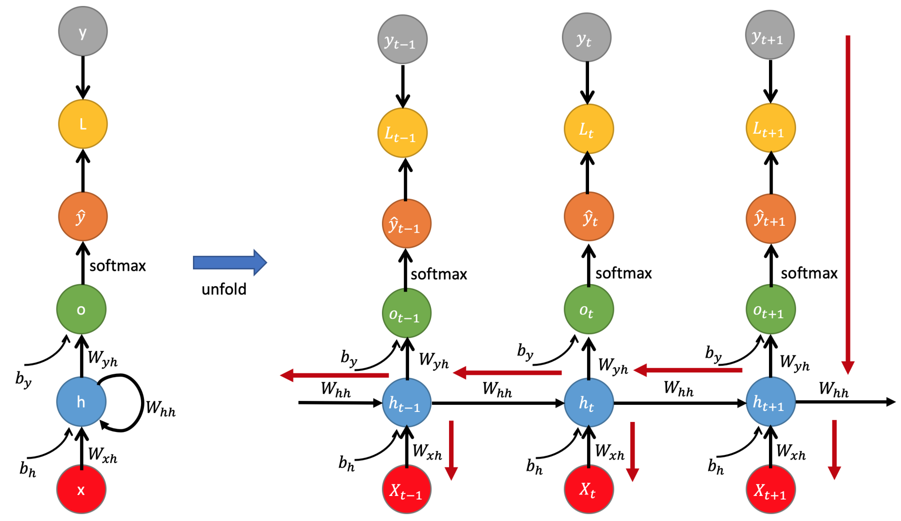


Figure 3.4 Unfolded RNN to illustrate the BPTT algorithm. Taken from `https://raw.githubusercontent.com/mmuratarat/mmuratarat.github.io/master/_posts/images/BPTT.png`.

FNN with the number of layers equal to the number of time steps. Furthermore, the total error of the network is then calculated as the sum of the errors for each time step.

So, the standard backpropagation algorithm shown in section 3.1.5 can be used for this unfolded network with the difference that the sum of gradients over all time steps is used as the subtraction term that was used in equation 3.11. For example, for a gradient with respect to W_{hh}, which are the weights between two connected hidden layers as shown in figure 3.4, a given subtraction term can be seen in equation 3.14 below [15].

$$\frac{\partial L}{\partial W_{hh}} = \sum_{t}^{T} \sum_{k=1}^{t} \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_{hh}}, \tag{3.14}$$

where h_t is a hidden state of a time step t and $\frac{\partial h_t}{\partial h_k}$ is calculated as follows:

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k}^{t} \frac{\partial h_{j+1}}{\partial h_j} = \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \dots \frac{\partial h_{k+1}}{\partial h_k}. \tag{3.15}$$

3.2.2 RNN limitations

RNNs should, in theory, be able to handle long-term dependencies. However, in reality, it turns out that specific problems arise when learning the network during the backpropagation. The problems occur when a given gap between the current input and a past input that somehow affects the current input is too large. In this case, RNNs are prone to problems with gradient. These problems are two, namely vanishing gradient problem and exploding gradient problem. Some papers point out that it only takes 8–10 hidden layers to cause the mentioned problems [12, 14, 15].

3.2.3 Gradient problems

Gradient problems appear during the actual learning of the RNN, which is done using the back-propagation through time algorithm. Backpropagation through time uses a gradient, specifically at the point when the individual weights are recalculated. According to [18], if a given sequence is too long, it may happen that “long term components go exponentially fast to norm 0, making it impossible for the model to learn correlation between temporally distant events”. The opposite can also happen, where long term components grow exponentially.

This is because the derivative of the vector function with respect to the vector is performed when calculating the gradient. Specifically, in equation 3.15 there is a multiplication of Jacobian matrices, the result of which is also a Jacobian matrix. Furthermore, it can be shown that if the highest eigenvalue of the resulting Jacobian matrix is less than zero, the vanishing gradient problem occurs. Further, for the vanishing gradient problem to occur, the highest eigenvalue of the matrix must be greater than 1. More can be read in [18].

3.3 Long short-term memory

The first model, which belongs to the RNN, is the Long short-term memory (LSTM). The LSTM has a similar structure to the RNN, except that each node in the RNN is replaced by a so-called memory cell, which I will discuss later. This model is able to learn long-term dependencies, which can be helpful, for example, in time series processing (video processing, audio processing), or another language modelling, machine translation, handwriting recognition, speech recognition, speech synthesis and text generation. The LSTM model was introduced by [19] and was designed in particular to deal with the Vanishing gradient problem and Exploding gradient problem that occur in recurrent neural networks [15].

3.3.1 Original version

In the original version of the LSTM, which Hochreiter and Schmidhuber proposed in 1997, each memory cell had in the core a CEC (Constant Error Carousel), which is a central linear unit that is recurrently self-connected. The activation of CEC is called the cell state. Thanks to CEC, the vanishing gradient problem can be solved by maintaining a constant error flow (weight set to a fixed value of 1). Furthermore, the memory cell is protected by the gates, which control what is changed in the memory cell. An input gate protects the memory cell from being modified by input data that is not relevant. Input gates also decide whether to leave the data in the memory cell or overwrite it. The output gate protects other units from irrelevant current memory states that could affect them negatively by filtering the output. It also decides when to access the current cell state. It is essential to have control over what information is being kept inside the memory cell, as it further interacts with other units. The memory cell can be seen as a bridge that can connect past information with current information, which makes the LSTM model so effective when dealing with time series or, in general, with data that are dependent on each other (translation, handwriting recognition, ...). Another advantage that memory cells offer is that the input gate (output gate) can use input from other memory cells to decide whether to store certain information in the current memory cell. However, this structure still has its shortcomings [19].

Although the CEC solves the vanishing gradient problem, there is another problem. If the data is too long and continuous, it can get to a point where the values in the memory cell grow so high that there is no control over them. The original design of the LSTM model assumed that a given input would have the data split so that the end of the sequence and the beginning of the next one would be recognized. Then, at the beginning of the new sequence, the values in the memory cell should be reset to zero. However, the problem appears if the input sequence is too long or not split into more minor sequences at all. This results in the values not being reset as often, causing the cell states to grow linearly with the length of the sequence, which in the case of a long sequence means indefinitely. This fact can result in saturation of the squashing function. Saturation will cause the squashing function’s (sigmoid) derivative to vanish [20].

This problem can be solved by introducing a new gate, called forget gate, instead of CEC.

This idea was introduced in 2000 by Felix Gers et al. [20]. Forget gates are used to reset memory cells. Thus, the memory cell does not reset only at the beginning of a new sequence as it was before, but the network learns itself when and what to reset. This change has proven to be effective over time and thus has become part of the standard implementation of the LSTM model [21].

3.3.2 Modern LSTM structure

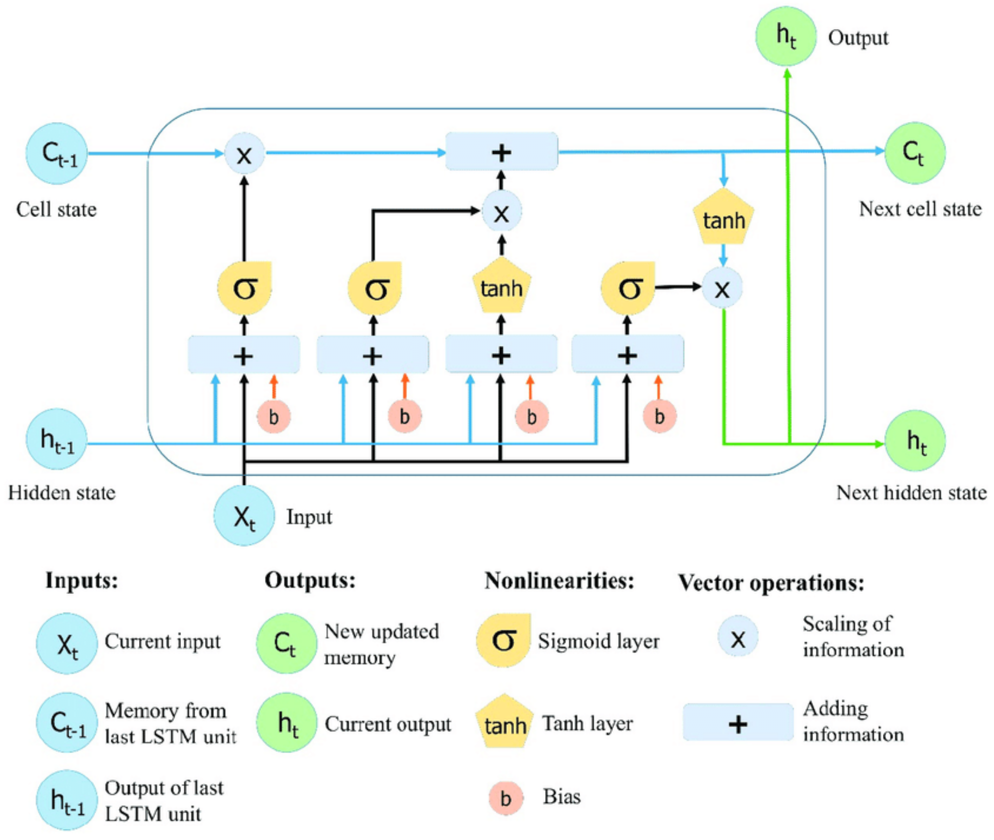


Figure 3.5 Long short-term memory unit. Taken from [11].

Modern LSTM consists of four parts, which can be divided into a cell and three gates. These gates are input, output and forget. All these parts are located in a structure named memory block. This structure can contain one or more memory cells or also called units, which are the main building blocks of the LSTM model because they can store information in the model. Furthermore, the structure contains a cell state, which can be thought of as a chain that runs through the entire structure. The information is added to it, removed and modified using the gates mentioned above, so the cell state serves as a kind of memory. This information can remain in the memory cell all the time, depending on its relevance. As part of the learning process, the network also learns what information to keep and what to throw away as time grows.

The individual gates contain functions called squashing functions, which are functions that expect an arbitrarily large input and return a number that is bounded from the top and also from the bottom. Specifically, the sigmoid function is used, which returns a number between 0 and 1. This number indicates how much information should be passed on. For example, for forget gate due to vector multiplication, 1 means leave the information as it is, while 0 means forget the information [14, 21, 22].

The whole structure of the LSTM unit can be seen in figure 3.5.

3.3.3 Forget gate

Firstly, the input passes through the forget gate, which determines which information should be erased from the cell state because it is no longer relevant. The actual deletion may not take place immediately, the information may gradually decrease until it reaches zero, at which point it is forgotten. The forget gate expects the hidden state of the previous LSTM unit h_{t-1} on its input, as well as the current input x_t. These values are further sent to the sigmoid function, which returns a number for each information in the cell state. The output of the sigmoid function f_t is further multiplied with the cell state from the previous LSTM unit C_{t-1} using a pointwise multiplication operation.

The final formula is as follows:

$$f_t = \sigma(W_f \cdot x_t + U_f \cdot h_{t-1} + b_f), \tag{3.16}$$

where W_f is vector of input weights, x_t is input at time t, U_f is vector of weights of forget gate, that is shared across all time steps, which means the forget gate weights are time-independent, h_{t-1} is previous hidden state, b_f is bias and σ is sigmoid function [14, 21].

3.3.4 Input gate

The next step is to update the values in the cell state and decide what information should be added. The values are updated using the input gate, which again receives the hidden state of the previous LSTM unit h_{t-1} and the current input x_t on its input. This is again sent to the sigmoid function. The formula is as follows:

$$i_t = \sigma(W_i \cdot x_t + U_i \cdot h_{t-1} + b_i), \tag{3.17}$$

where W_i are input weights, x_t input at time t, U_i are input gate's weights, h_{t-1} is previous hidden state and b_f is bias.

To get new values that should be added to the cell state, the same inputs that were passed into the sigmoid function are taken and, this time, sent to the tanh function (hyperbolic tangent), which returns the weight between $[-1, 1]$. This number indicates the "level of importance". This will create a vector $\widetilde{C}_t$ of new candidate values that should be added to the cell state. To calculate new values $\widetilde{C}_t$, the following formula is used:

$$\widetilde{C}_t = \tanh(W_c \cdot x_t + U_c \cdot h_{t-1} + b_c), \tag{3.18}$$

where W_c are input weights, x_t input at time t, U_c are weights which regulates new values, h_{t-1} is previous hidden state, b_c is bias and tanh is a hyperbolic tangent function.

At this point, the model has everything that it needs to modify the cell state C_t at time t. So firstly, the network does the forget phase and then the update phase. The input gate also serves as a sort of shield against adding information to the cell state that is irrelevant and could potentially negatively affect following units. The current cell state C_t is updated as follows:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \widetilde{C}_t, \tag{3.19}$$

where $\widetilde{C}_t$ is the vector of candidate values, C_{t-1} is previous cell state, i_t is input gate and f_t is forget gate [11, 14, 15, 22, 23].

3.3.5 Output gate

The next and final stage that the LSTM model does is to decide what values should be at the output of the current memory cell. The decision is made by the sigmoid function and is calculated as follows:

$$o_t = \sigma(W_o \cdot x_t + U_o \cdot h_{t-1} + b_o), \tag{3.20}$$

where W_o are input weights, x_t is input at time t, U_o are weights of output gate, b_o is bias and σ is sigmoid function.

This output o_t is further multiplied by the cell state C_t of the current memory cell, which is inserted into the tanh function before the actual multiplication. The output of the tanh function is therefore multiplied by the output of the sigmoid function as can be seen in the the following formula:

$$h_t = o_t \cdot \tanh(C_t), \tag{3.21}$$

where o_t is output gate, C_t is cell state at time t and tanh is a hyperbolic tangent function.

The output gate decides what should be in the final output of the current unit. The resulting hidden state h_t is sent as the output of the current memory cell (if the user wants it that way). It is also sent to the next memory cell as a hidden state h_t along with the memory state C_t. Thanks to the output gate, the model does not output the entire cell state but only the information that the model wants to. This can protect other units from irrelevant information or any kinds of disturbances [11, 14, 15, 22].

3.4 LSTM variants

Since the LSTM has been around for quite a long time, many variants have been created. I have already shown the first variant, the original LSTM with forget gate. Another variant was even created in the same year when Felix Gers came up with the idea of the forget gate. That year he and the author of the original model Schmidhuber came up with a variant of the peephole LSTM. This variation has so-called Peehole connections, which are used to allow each gate to access the internal state of the memory cell. Gers et al. [24] mention that “this makes LSTM a promising approach for numerous real-world tasks that require to learn to measure the size of time intervals” [14, 15, 24].

The last variant worth mentioning is the GRU, which I will discuss separately.

3.5 Gated recurrent unit

Another model that can be included in the RNN is the Gated recurrent unit (GRU). The model was introduced in 2014 by Cho et al. [26]. This model is similar to the LSTM model and can be said to be its more modern equivalent. This variant of the LSTM is a bit more different than the other variants. The main difference is that the cell state has been removed. It has been merged together with the hidden state. Also, the output gate itself has been removed.

Another modification is merging the forget and input gates into a single gate called the update gate. This gate is used to do both of the things that the original input and forget gates did, i.e. it decides what information should be added to be remembered and what information should be thrown away. Another gate that can be found in the GRU model is the reset gate, which is used to let the network decide how old information should be discarded. Due to the smaller number of gates, the GRU model has fewer parameters than the LSTM and is, therefore, easier to learn in terms of training time.

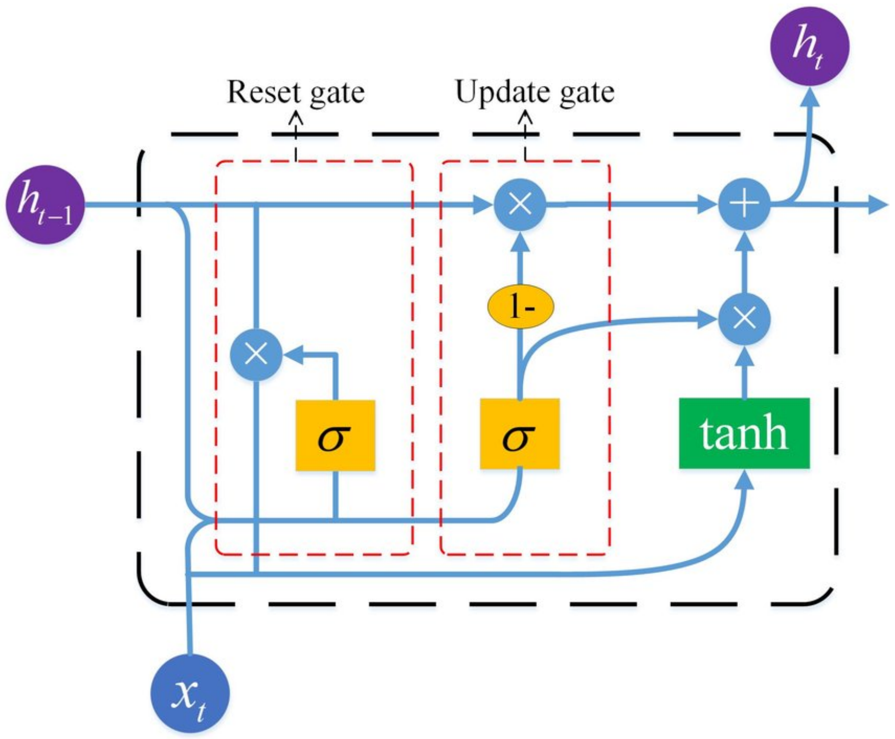


Figure 3.6 Gated recurrent unit. Taken from [25].

The disadvantage of the GRU model is that it always exposes the entire memory. There is no way to limit it as the LSTM did with the output gate. Many papers also point out that GRU gives better results in almost all RNN application areas. The whole structure of the GRU unit can be seen in figure 3.6 [14, 27].

The activation h_t of the gated recurrent unit at time t can be calculated as follows:

$$h_t = (1 - z_t) \cdot h_{t-1} + z_t \cdot \tilde{h}_t, \tag{3.22}$$

where h_{t-1} is previous hidden state, z_t is update gate, which regulates what and how much should be updated and $\tilde{h}_t$ is candidate activation.

Update gate z_t can be computed by:

$$z_t = \sigma(W_z \cdot x_t + U_z \cdot h_{t-1}), \tag{3.23}$$

where σ is sigmoid function, W_z are input weights, U_z are update gate's weights, x_t is input at time t and h_{t-1} is previous hidden state.

As you can see, the equations for update gate in gated recurrent unit are similar to those used in input (3.17) and forget (3.16) gate in the LSTM model.

The candidate activation $\tilde{h}_t$ is calculated as follows:

$$\tilde{h}_t = \tanh(W \cdot x_t + U \cdot (r_t \odot h_{t-1})), \tag{3.24}$$

where r_t is a reset gate, $\odot$ is element-wise multiplication, W are input weights, x_t is input at time t, U are weights that control new values and h_{t-1} is previous hidden state..

The equation for the reset gate r_t is as follows:

$$r_t = \sigma(W_r \cdot x_r + U_r \cdot h_{t-1}), \tag{3.25}$$

which is similar to the update gate. The only difference is that the used weights are weights of reset gate [27].

3.6 Stacked or Unidirectional architectures

In their work, Cui et al. [28] mentioned that existing articles “have shown that deep LSTM architectures with several hidden layers can build up a progressively higher level of representations of sequence data, and thus, work more effectively”. This statement can be taken from a greater perspective, which means that a similar approach could also be used to improve RNN models and ANNs in general by having several hidden layers stacked on top of each other. Furthermore, the output of one layer is passed as input into the next layer. The resulting model is called Stacked or Unidirected architecture. This combination of layers can result in the model returning more sophisticated data patterns. The name of the architecture implies that all stacked layers process the sequence in the same flow direction. This means that they cannot see into the future and cannot learn dependencies from both directions. Thus, they learn only from previous data [29].

3.6.1 Stacked LSTM, Stacked GRU

Instead of RNN, the above-mentioned models, such as LSTM and GRU, can be used. Created models are called Unidirectional Stacked LSTM and Undirectional Stacked GRU. The structure of the Stacked LSTM model can be seen in figure 7. In a way, it can be said that a Bidirectional architecture (which will be discussed in the next section) is a subgroup of a stacked architecture since both models have stacked layers. Still, as the name suggests, this structure is bidirectional.

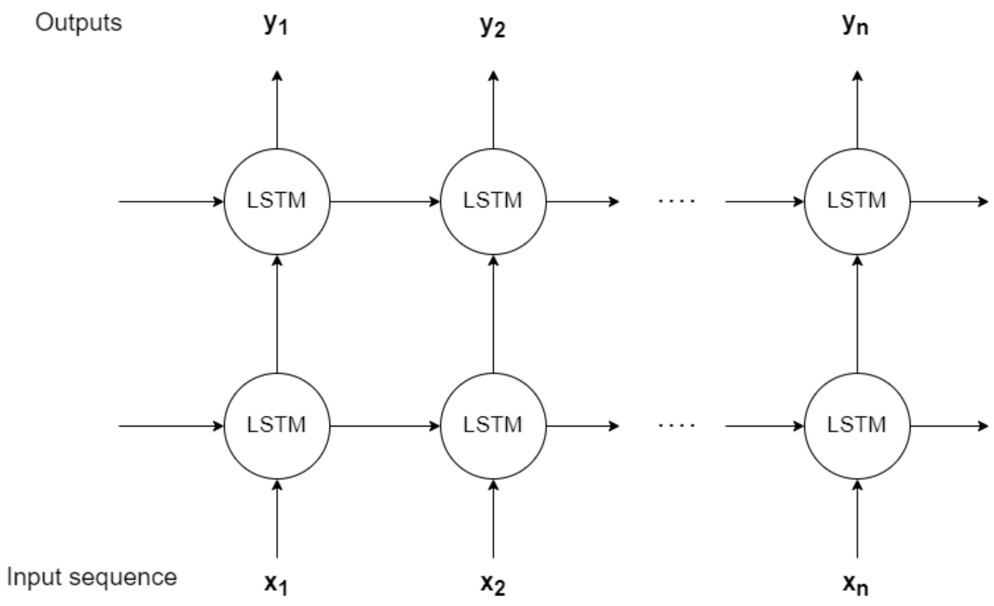


Figure 3.7 Stacked architecture with two LSTM layers.

3.7 Bidirectional architectures

So far, I have introduced models that had a “casual” structure. This structure’s output at time t was affected only by the inputs and internal states from the past, i.e. $x_1, ..., x_{t-1}$ and the current input x_t. However, some problems suggest that it might not be a bad idea for the model to have access to the entire sequence and hence to future data. For example, when solving the problem of filling a word in a sentence within a context, the model might want to know words which come after the word that the model is currently supposed to fill in. Other problems where the above approach could be applied are handwriting recognition, translation and many others [30].

The first ideas about how the model could look at future information were that the RNN would not give a result at time t, but would return the result after M other time blocks. This would delay its output and allow it to look at other future data. Theoretically, the proposed method could work. However, in reality, it turns out that the accuracy of the result decreases with a higher time delay M. To take into account the whole sequence, or at least a more significant subset of an input sequence, a different approach needs to be invented. A method that would satisfy the desired conditions should use all the information.

This can be achieved by making two separate networks. Each of these networks will have the opposite flow direction so that one will take the sequence from the beginning, and the other will take the sequence from the end. The result is then a combination of both directions, which would give the model information from both the past and the future data. There is still one problem though, and that is how to connect both directions.

One approach is more analytical and statistical. In this case, one takes the two directions as separate experts, then does their merging using the arithmetic mean for regression and the geometric mean for classification. However, this approach assumes the independence of the experts. In practice, it turns out that this approach is not entirely appropriate since the two different directions (experts) are not independent. This stems from the fact that the various networks are learned on the same data [31, 32].

3.7.1 Bidirectional RNN

Due to the limitation of conventional RNN, bidirectional recurrent neural networks (BRNN) were designed. The principle is the same as above. The network is divided into two RNN subnetworks, each with opposite directions. The Bidirectional architecture is similar to the previously mentioned Stacked architecture. The individual layers are also stacked, but with the difference that the direction flow is opposite in each layer. The outputs of each layer can be calculated according to the equations below.

For a forward hidden layer h_t^F:

$$h_t^F = \sigma(W_x^F \cdot x_t + W_h^F \cdot h_{t-1}^F + b_h), \tag{3.26}$$

where x_t is input at time t, h_{t-1}^F is previous hidden state of forward layer at time $t-1$, W_x^F and W_h^F are weights in forward pass for input x_t and hidden state h_{t-1}^F respectively and b_h is bias.

For a backward hidden layer h_t^F:

$$h_t^B = \sigma(W_x^B \cdot x_t + W_h^B \cdot h_{t+1}^B + b_b), \tag{3.27}$$

where x_t is input at time t, h_{t+1}^B is previous hidden state of backward layer at time $t+1$, W_x^B and W_h^B are weights in backward pass for input x_t and hidden state h_{t+1}^B respectively and b_b is bias.

The final result is further calculated using the following formula:

$$y_t = h(W_y^F \cdot h_t^F + W_y^B \cdot h_t^B + b_y), \tag{3.28}$$

where h_t^F is hidden state of a forward layer at time t, h_t^B is a hidden state of a backward layer at time t, W_y^F and W_y^B are weights for hidden layers for a forward and a backward layer respectively and b_y is bias.

Since the network tries to minimize the loss function for both directions simultaneously, there is no need to deal with how to connect the individual directions [32].

The basic idea of Bidirectional RNN can be extended to two dimensions, which could be used for images, for example. In this case, the network would not consist of two but four RNNs for each direction - up, down, left and right [30].

3.7.2 Bidirectional LSTM, Bidirectional GRU

Bidirectional LSTM can be seen as a Bidirectional RNN. The only change is that there are LSTM memory blocks instead of hidden layer activation. The whole structure of Bidirectional LSTM model can be seen in figure 3.8. In the case of the GRU model, GRU units are used instead of hidden layer activation [33].

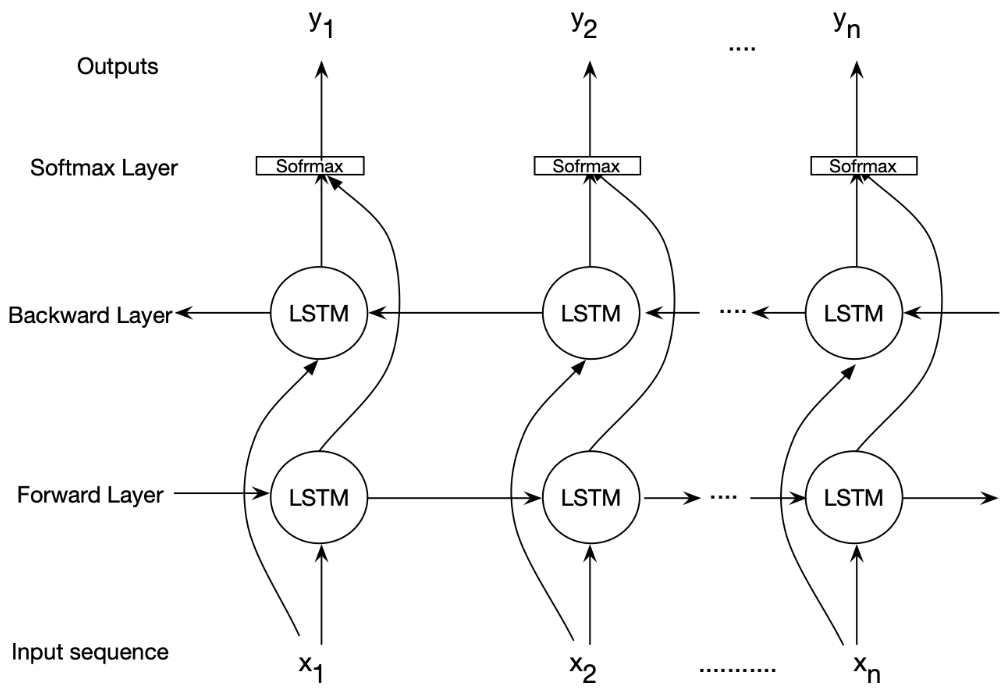


Figure 3.8 Bidirectional architecture with two LSTM layers, each layer has an opposite flow direction with the Softmax layer. Taken from `https://nedatavakoli.github.io/Modeling-and-Clustering-Genome-using-Bidirectional-LSTM/CS6220-ModeilingGenome/Architecture/BiLSTM.png`.

3.8 Encoder-Decoder

This type of architecture was invented by Cho et al. [34]. The architecture is very efficient because it can map a sequence to another sequence. The advantage is that the input and output sequences do not necessarily have the same length. This fact is used, for example, in translation, where an input sentence in one language may have a different word count than an output sentence in another language. The resulting model consists of two submodels. As the name implies, these are an encoder and a decoder.

3.8.1 Encoder

The encoder is used to read and encode the input and create a so-called context C that represents and describes the input sequence. This context is a vector of fixed length. The encoder usually consists of an RNN that reads the sequence sequentially and uses the last hidden state as the context C.

3.8.2 Decoder

The decoder is usually also made up of an RNN and is used to create a sequence from a given context C. This context serves as input to all RNN units. Thus, the current hidden state h_t of the decoder at time t is determined by the previous hidden state h_{t-1}, the previous output y_{t-1} and context C and is computed by:

$$h_t = f(h_{t-1}, y_{t-1}, c), \tag{3.29}$$

where f is an activation function [30, 35, 34].

3.9 Convolutional neural network

The convolutional neural network (CNN) was created in 1995 by LeCun et al. [36]. The greatest strength of CNN is that it can extract features. The structure of CNN consists of five layers laid one behind the other. These layers are called input, convolutional, pooling, fully connected and output. The network is called convolutional because it uses convolution, which is a mathematical operation on two functions. A common use of CNNs is for tasks where the input data are in the form of a grid. So the usual use is for images, but a time series can also be thought of as a 1D grid. Thus, there is a possibility to use CNNs also for time series.

Another reason why CNN is primarily used for images is that if the image were to be processed using ANN, there would be many parameters to be set. For example, for an image size of 64×64, there would be a need to deal with 12 288 weights on one neuron (one weight for each pixel). CNN uses so-called filters or kernels. Moreover, these kernels share the weights for each point of the input matrix, thus rapidly reducing the number of network parameters, which made ordinary ANNs a problem [37, 9, 38, 39, 40].

3.9.1 Convolutional layer

The first layer I am going to talk about is the convolutional layer. This layer is used to extract features and performs the main part of the whole network, the convolution operation. The output of the convolution is then mostly sent to the activation function, which is nonlinear. Thus, a linear (convolution) and nonlinear (activation function) combination are obtained. The most well-known activation function is the ReLU function [39].

3.9.1.1 Convolution

Convolution is a mathematical operation that is used for feature extraction. It is a specialized kind of linear operation. As I wrote earlier, convolution handles two functions. In CNN, it can be understood that the input is one function and the individual kernels are the other functions. Convolution is then the operation that determines the changes of the input caused by the kernel. Let's have an input matrix V of size $N \times N$ and the filter W with $F \times F$ size. The input changes are computed according to the following formula:

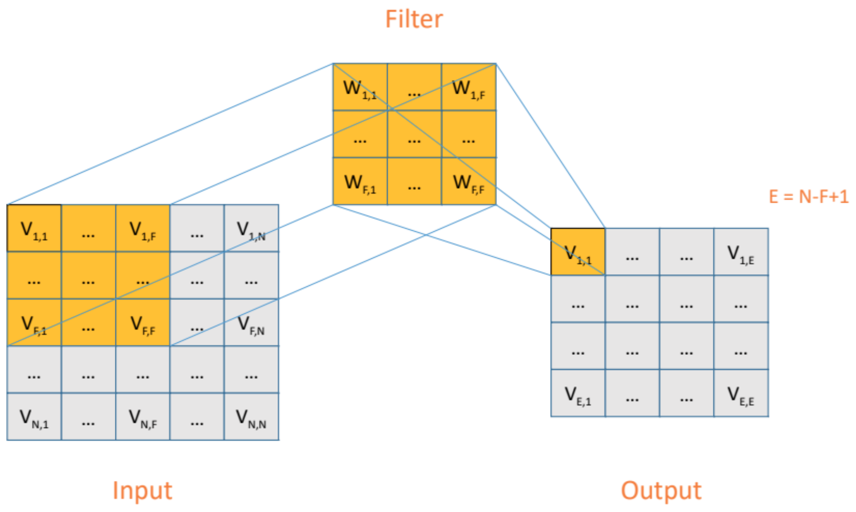


Figure 3.9 Application of the convolution on the first position with a filter of size $F \times F$ to get value of $V_{1,1}$. Taken from [5].

$$v_{i,j} = \sum_{k=0}^{F-1} \sum_{m=0}^{F-1} w_{k,m} V_{i+k,j+m}. \tag{3.30}$$

In figure 3.9, you can see the kernel applied to the input to get the value $v_{1,1}$. This is repeated for all points in the input matrix, assuming that stride equals 1. (I will discuss what stride is later). The resulting matrix, which comes out by applying the filter to the input matrix, is called a feature map. Before the new feature map itself is sent on to the next layer, it usually goes through an activation function.

As you can see from the figure 3.9, the resulting feature map has a smaller size than the input matrix. The specific sizes of the output are $N - F + 1$. This shrinkage can be avoided by adding so-called padding to the input matrix [37, 39].

3.9.1.2 Padding

Padding is a technique to affect the output size of the convolutional layer. Padding works by adding zeros from each side of the input matrix. Enough zeros needs to be added to keep the feature map dimensions the same as the input. The way to do this is that when the output is computed for the first point $v_{1,1}$, the centre of a kernel will sit on that point. For example, for a 3×3 kernel, we only add one “layer” of zeros [39].

3.9.1.3 Stride

A stride is the size of a kernel’s shift. In typical cases, the stride equals 1, but sometimes it is better to choose a larger stride in case of the need for downsampling of the feature maps because if the kernel moves more squares, there will be fewer results in the feature maps [39].

3.9.2 Pooling layer

The pooling layer is primarily used for downsampling feature maps. This reduces the complexity of the network and, therefore, the learning time. Due to the reduction in network parameters, CNNs are also quite resistant to overfitting. The pooling layers also contain a kernel, usually of size 2×2 and stride set to 2. This reduces the input matrix size by a factor of 2 (a 4×4 matrix at the input is reduced to 2×2), so overall, the number of parameters is reduced to 25 % of the original size. Pooling works by taking all the values from the feature maps inside the kernel and making a single number out of them. The most well-known forms of pooling are Max pooling and Average pooling [37, 40].

3.9.2.1 Max pooling

Max pooling is probably the most popular form of pooling. It works by taking the maximum value of all the values in the region represented by the filter [39].

Let's say the size of the kernel is $F \times F$ and input matrix is x. Max pooling can be computed as follows:

$$y_{i,k} = \max_{k=0,\dots,F,m=0,\dots,F} x_{i+k,j+m}. \tag{3.31}$$

3.9.2.2 Average pooling

Another type of pooling is Average pooling, which works in the same way as the previous Max pooling, but returns the average of the values instead of the maximum [39].

3.9.2.3 Global average pooling

Another type of pooling worth mentioning is Global average pooling. This method works by taking the average of each feature map. So it makes only one number out of each feature map. Global average pooling is also able to reduce the number of parameters, and also, thanks to this, the input can be of any size. So it does not necessarily have to be a square matrix [39].

3.9.3 Fully connected layer

The next layer is called the fully connected layer. It is usually placed either after the convolutional or pooling layer. The output of these layers is flattened, so the output is changed into a vector. This vector is sent to the fully connected layer, also called the dense layer. The layer has connected neurons so that all previous layer neurons are connected to all neurons in the fully connected layer. The output number of neurons is the same as the required number of classes for classification. Then the output of this layer is further sent to an activation function, usually ReLU, or softmax [39].

Let's assume that v_i^j is the value of i neuron in j layer and $w_{k,i}$ are connection's weights. The output of neuron v_i^j is calculated from all neurons in the previous layer k. The formula is as follows:

$$v_i^j = \sigma\Big(\sum_k v_k^{j-1} w_{k,i}^{j-1}\Big). \tag{3.32}$$

3.9.4 The overall structure of CNN

In figure 3.10, you can see an example of a CNN. The overall structure of CNN can vary a lot, and it primarily depends on the problem it is supposed to solve. The network consists of the layers mentioned above. However, the layers may be repeated. For example, sometimes, it is convenient to use a convolutional layer again after the pooling layer. This convolutional layer can be followed by the pooling layer again and so on. Another possibility is, for example, to use

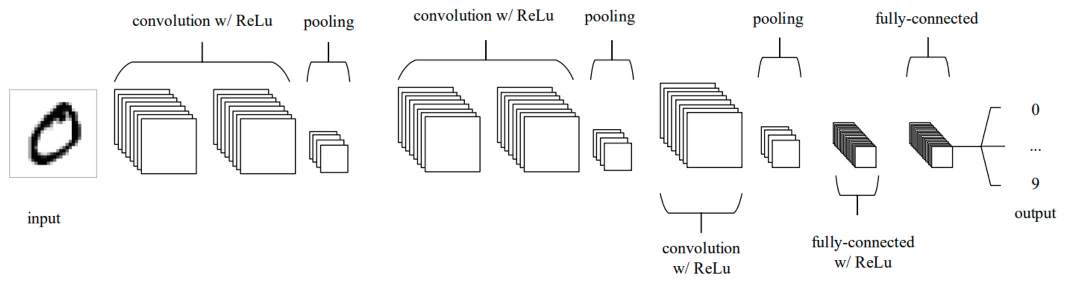


Figure 3.10 Example of CNN architecture, which consists of Input, convolutional, pooling and fully-connected layer, stacked in specific order. Taken from [40].

two convolutional layers directly behind each other, followed by a pooling layer. There are many options, and it is not possible to determine which one would be universally the best [38, 40].

3.10 CNN-LSTM

The last model I will talk about is the CNN-LSTM model. The model can be seen in figure 3.11, and it is a combination of the CNN and the LSTM models. The resulting model should benefit from the strengths of both models. Convolutional layers are suitable for feature extraction, while the LSTM is good for capturing dependencies in the data. The final model should have both strengths [41].

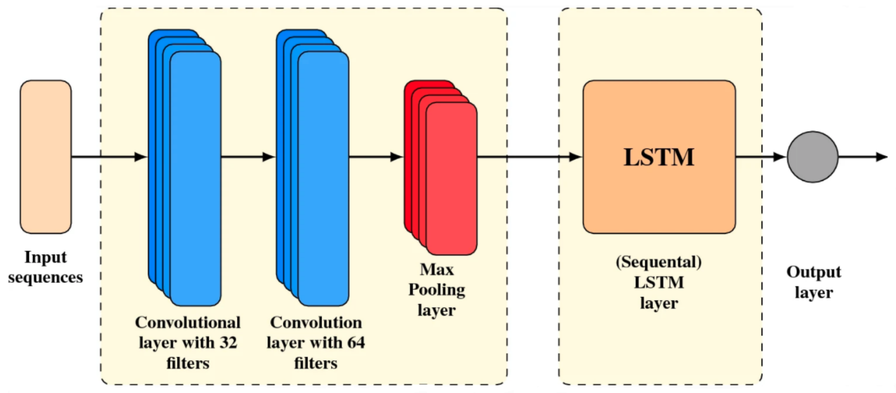


Figure 3.11 Example of CNN-LSTM model with two convolutional layers, one max pooling layer and one LSTM layer. Taken from [41].

3.11 Chapter summary

This chapter explained and introduced the basic models that can be applied to stock market forecasting. Furthermore, neural networks and recurrent neural networks were explained in order to present more complex models afterwards. The models mentioned include LSTM, GRU, CNN, CNN-LSTM, Encoder-Decoder architecture, Bidirectional architecture and Stacked architecture. The chapter also contains the RMSE metric that I will use to evaluate the models.

Chapter 4

Implementation

The chapter describes the basic implementation details. It defines what programming language and libraries were used in the thesis and why these tools were chosen. Then the layers used and their implementation details are described.

For the implementation, I chose the Python programming language because it is the most widely used language for ML and also has the most support for creating neural networks from my point of view. Another advantage is the implementation itself in Jupyter notebooks, which is a development environment in which the Python code can be executed. Furthermore, the Jupyter notebooks can be used in Google Colaboratory, which I will talk about separately.

4.1 Keras and TensorFlow

Keras is an Open Source Neural Network library written in Python. Keras was created for ease of use and simplicity when building neural networks and ML models. The Keras library runs on TensorFlow, a Python library that has the advantage of being GPU and CPU optimized. TensorFlow was created by Google and is used to solve machine learning tasks.

In this work, I used layers from the Keras library, which are described below. The layers that will be mentioned are further used to create models proposed in chapter 7.

4.1.1 LSTM and GRU layer

The LSTM layer provided by the Keras library implements the base version of the model, so it is not a peephole variation. The main parameters of the unit include the number of hidden neurons. This parameter is the only mandatory one. After that, the user has the possibility to change a large number of other parameters, which are set by default. These parameters include *dropout*, *recurrent dropout*, *activation*, *recurrent activation* and many more. The last parameter I would like to mention is *return_sequences*. This parameter is used to determine whether each LSTM unit should return its result (the output of the entire model is then a sequence of results) or if only the result from the last unit should be returned. I modify this parameter in my thesis as needed. For example, for the Stacked approach, I require that a specific unit at time t sends its result to the next unit in the next layer, which is also at time t. Thus, I need each unit to send its output, which I achieve by setting the *return_sequences* parameter to *True*.

The next layer that has been used in this work is GRU, which is very similar to the previous one. It contains similar parameters so I will not comment too much on it.

4.1.2 CNN layers

The following layers that I will mention were used in the CNN model. These are the Conv1D, MaxPooling, Flatten and Dense layers. I use the Dense layer in other models (e.g. LSTM), and it serves primarily as an output layer. The other layers that have been used in the CNN models do what the name implies. Conv1D performs the convolution operation. The kernel in this layer is one-dimensional since it accepts vectors at the input layer. The mandatory parameters for this network are *filters* and *kernel_size*, which are used to determine the number of filters and the kernel's size. In addition, the user can set *padding* and *strides*. The MaxPooling layer performs the standard Max Pooling operation, explained in section 3.9.2.1. The last layer is the Flatten layer, which shrinks the input to a vector. Thus, the size of the vector is the product of the individual dimensions.

4.1.3 Bidirectional layer

This layer is used to create the Bidirectional architecture described in section 3.7.1. The Keras implementation accepts the model as a parameter and also other optional parameters. These include the *backward_layer*, which is the backward model. This parameter is optional, as already mentioned. If the user does not fill in the parameter, Keras uses the model that was passed to the layer as the *layer* parameter, which is the original forward model. This implementation also requires that the inserted models have the values of their *go_backwards* parameters set in reverse.

4.1.4 Encoder-Decoder layers

The layers that have been used in the Encoder-Decoder model include TimeDistributed and RepeatVector. The RepeatVector has n as a parameter, which indicates how many times the vector, or input in general, that the layer receives should be repeated. So if the input has size *(num_samples, features)*, the output will have size *(num_samples, n, features)*, where *num_samples* is a number of input data and *features* is the number of features provided to models. The TimeDistributed layer is used to apply some other layer, which it takes as a parameter, to every temporal slice of input data. The layer requires its input to be at least three dimensional.

4.2 Google Colaboratory

Google Colaboratory is a product from Google Research. It is primarily used to run Python code in the browser. The advantages are that the user does not have to worry about the virtual environment. So, for example, when sharing a Jupyter notebook, the author does not have to worry about anything. Another advantage is the optimization mentioned above of the Keras library for GPUs. Google Colaboratory offers to switch the runtime to run on the GPU, but this feature is time-limited. Using the GPU, I trained the models approximately ten times faster than on a local machine. Google Collaboratory can also connect to Google Drive, which can be suitable for saving individual models. Last but not least advantage of Google Colaboratory is that the program runs on a different machine than the user's pc, so the user does not have to use his resources.

4.3 Chapter summary

This chapter introduced the programming language in which the models are implemented. Furthermore, the Keras library in which I created the selected models is presented. Then the layers that were used to create the models are described. Also, the chapter mentioned the Google Collaboratory tool that was used to run the Python code.

Chapter 5
Data

This chapter introduces the stock market that has been selected for learning and evaluating models. It also describes how the data was collected. The chapter includes a basic description of the dataset along with a statistical description of the selected target column. Next, the data splitting style is introduced, followed by a description of the types of predictions for which the data must be specially prepared. Subsequently, the data preprocessing flow is described.

5.1 Data selection

As I mentioned before, for learning and validating each model, I will use SXR8.DE data, which is an ETF (Exchange-traded fund) that replicates the S&P 500 index, one of the most famous indexes in the world. It is an index that contains the 500 wealthiest companies in America. I generally chose this ETF and this index because it is one of the most well-known. Many investors consider it the least controversial because it has continued to grow historically. Therefore they claim that for people who do not have experience with trading on the stock exchange, investing in the S&P 500 is the easiest option.

5.2 Data collection

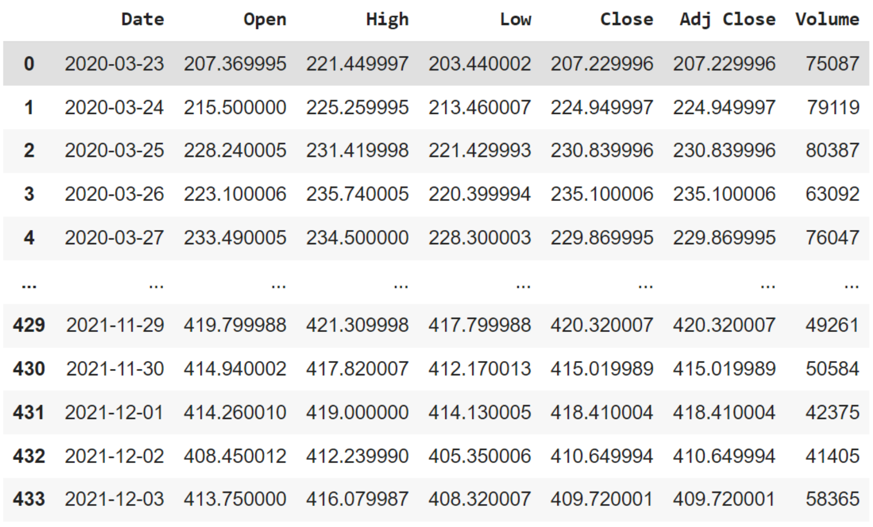
The data has been collected on the Yahoo Finance[1] website, which is a website where anyone can look at individual stock markets. There is also news regarding the stock markets, which an investor can also use to make a prediction or to determine whether a stock will rise or fall. Yahoo Finance also offers a basic price chart as well as historical data that goes back to the beginning of the selected stock market. So on Yahoo finance, all you have to do is select the stock market, then choose the period from which you want the data, and then click the download button to download the data in CSV format, which is a standard format for people who work with data, so you do not have to convert the resulting dataset to another format.

5.3 Data description

The data I used was collected over the period March 23, 2020–December 3, 2021. The main reason for this choice of data was that I wanted to remove the dates when the COVID-19 pandemic started. Furthermore, before Christmas last year, the market began to be very volatile due to the situation in Ukraine. Since the models only rely on time series data, I wanted to minimize

[1]`https://finance.yahoo.com/`

these fluctuations. Since the stock exchange is open only on working days, the individual records are days of the week, excluding the weekends and holidays. The basic downloaded data contains 7 columns: “Date”, “Open”, “High”, “Low”, “Close”, “Adj Close” and “Volume”. The “Date” column describes what date it is. “Open” indicates the price at the stock exchange opening on a particular day. “High” indicates the highest price of the day, and “Low” is the lowest. “Adj Close” or adjusted close indicates the closing price after adjusting for all relevant splits and dividend payments. The last column is “Volume”, which indicates the number of shares of a company or index traded on a given day. The column that the models will try to predict is called “Close”. You can see a data example in figure 5.1.



	Date	Open	High	Low	Close	Adj Close	Volume
0	2020-03-23	207.369995	221.449997	203.440002	207.229996	207.229996	75087
1	2020-03-24	215.500000	225.259995	213.460007	224.949997	224.949997	79119
2	2020-03-25	228.240005	231.419998	221.429993	230.839996	230.839996	80387
3	2020-03-26	223.100006	235.740005	220.399994	235.100006	235.100006	63092
4	2020-03-27	233.490005	234.500000	228.300003	229.869995	229.869995	76047
...	...	...	...	...	...	...	...
429	2021-11-29	419.799988	421.309998	417.799988	420.320007	420.320007	49261
430	2021-11-30	414.940002	417.820007	412.170013	415.019989	415.019989	50584
431	2021-12-01	414.260010	419.000000	414.130005	418.410004	418.410004	42375
432	2021-12-02	408.450012	412.239990	405.350006	410.649994	410.649994	41405
433	2021-12-03	413.750000	416.079987	408.320007	409.720001	409.720001	58365

Figure 5.1 Example of data selected from Yahoo Finance. The first column denotes the date, the next column denotes the opening price of a particular day. Next are the columns with the highest and lowest value per day. Then there is a column that shows the closing price after adjusting for all relevant splits and dividend payments and the last column indicates the number of shares of a company or index traded on a given day.

5.3.1 Statistical description of the “Close” column

In image 5.2, you can see a Boxplot that shows how the data is spread out. The Boxplot shows half of the data, namely the data bounded by the 3rd quartile and the 1st quartile. These data are placed in a rectangle that also shows the median. The Boxplot also contains so-called whiskers, which indicate the range of the values. Sometimes you may also see a Boxplot with outliers behind the whiskers. However, there are no outliers in the dataset used in this work.

The median of the “Close” column is equal to 320.07. Further, the mean is 326.407 and the standard deviation is equal to 48.122.

Also, in figure 5.3 are shown the values of the “Close” column, which indicate the daily values of the observed dataset.

5.4 Data splitting

The next step after data collection is to split the data into training, validation and testing in the ratio 70:20:10. This approach is common in machine learning. The training part of the data is used to train the model. The validation is used to tune the hyperparameters, and the final testing is used to test the model on data that has never been seen to determine how good the

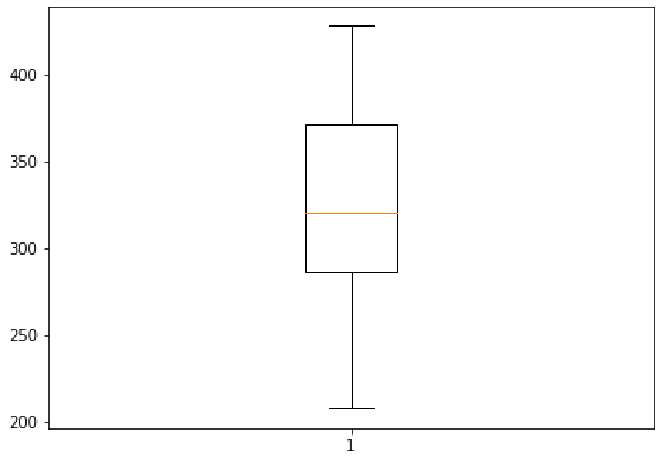


Figure 5.2 Boxplot of "Close" values of the dataset. The plot shows half of all the data. The chosen data is in the rectangle, which also contains the median. The data are bounded by the first and third quartile. Then there are whiskers to indicate the range of the data.

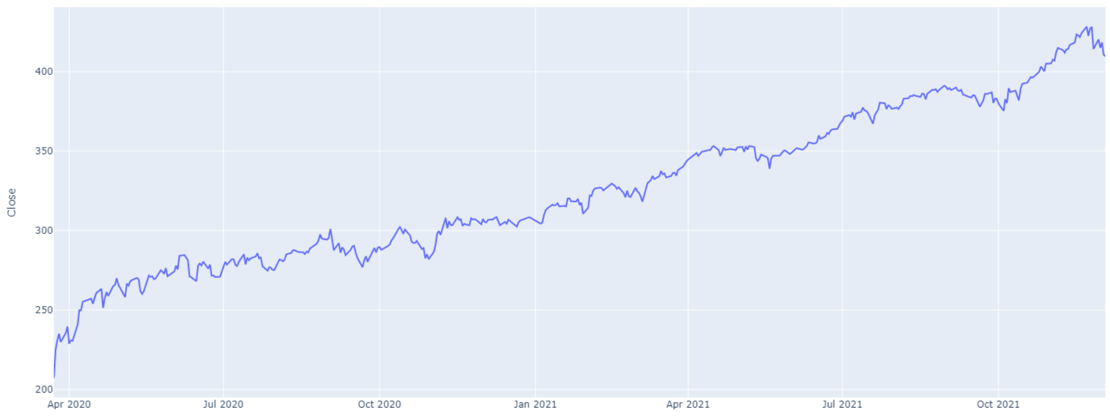


Figure 5.3 Graph showing "Close" column values of dataset. The chart shows the individual days on the X-axis and the values of the "Close" column on the Y-axis.

model will be for real-world usage. Since my data is in the form of a time series, I cannot use the usual techniques for data splitting. A function called *train_test_split()* from the scikit-learn library is often used to split the data. However, the splitting by scikit-learn is done randomly, which cannot be applied to a time series since the data is dependent on each other, so it cannot be randomly split that easily. Therefore, to split the data, I first calculate how much data will be in each part and then split all the data using Python slices, which preserves the dependency of the data between time points.

5.5 Division by type of prediction

The model can predict how many days ahead it wants, and it is primarily up to the individual user. In the real world, investors try to forecast as many days ahead as possible to maximize their profits. However, forecasting several days is much more challenging than predicting 1 day ahead. Therefore, I try to forecast 1-day and 5-day predictions in my thesis.

5.5.1 Prediction of 1 day ahead

This task is considered in most papers, and the reason is that the results are pretty promising, or at least they look good on a graph, which is also demonstrated in section 9.1. However, they still have their shortcomings, and later in the thesis, I will show that this prediction is not applicable in practice as the model overfits and does not predict well. Let us denote the number of days that the model will use to predict the next day as *time_steps* and the last day from which the following day is predicted as t. Then the prediction can be written as follows:

$$[t - time\_steps, ..., t - 1, t] \rightarrow [t + 1] \tag{5.1}$$

This is also what the final modified data should look like for one particular prediction of 1 day ahead, where the part before the arrow shows the input and the part after the arrow indicates the output or a forecast.

5.5.2 Multi-step prediction

This task is more complicated than predicting 1 day ahead, and this is because no one can say in advance how a given stock will move, as some significant change can happen that can significantly affect the development (the outbreak of the COVID-19 pandemic or a war). No one can expect these influences and changes, so in this thesis, I will not be dealing with an entirely long term prediction, but only for 5 days, which is a week in my dataset. The notation is the same as in the previous case. However, there is a need to denote the number of days that the model should predict. Let us denote this as n. The prediction for a specific point can be calculated as follows:

$$[t - time\_steps, ..., t - 1, t] \rightarrow [t + 1, ..., t + n] \tag{5.2}$$

Where again, the part before the arrow indicates the input data for prediction several days ahead, and the part after the arrow indicates the output or predicted data.

5.6 Data preprocessing

5.6.1 Separation of tasks

Since the data is already collected, the next step is to preprocess the data and prepare it to a state that can be used for training and validation. That being said, in this thesis, I am dealing with both 1-day and multi-step forecasting. Another division is how many columns from the original dataset will be used for prediction. The first option is to use only the “Close” column. In this case, the input is called univariate. If more columns are used, the input is called multivariate. The last division is the number of days, which the models will use to predict. The fundamental split will be 10 and 15 days, which means two weeks and three weeks. The data must be prepared separately for each type of problem as the input sizes differ. The next step is to prepare the input and output data.

5.6.2 Preprocessing of input data

The first step is to prepare the input data for the models. As I said, the original dataset has seven columns. However, some of them can be removed in advance as they will not play any role in the final prediction.

The first column that has no effect on the result and which I can therefore safely remove is the “Date” column, which specifies a particular date of the data.

Next, the input dimensions for the problems need to be determined. In the case of a prediction where multiple variables are used, the input will be of shape *(samples, time_steps, features)*, where *features* denote the number of variables (in the current case, features equals 5). The *time_steps* variable expresses what it did in the previous section, i.e. the number of days the model uses for prediction (specifically, the value will be equal to 10 and 15). Variable n describes how many days the model should predict, and variable *samples* represents the number of input data. For prediction using a single variable ("Close"), the input will be of shape *(samples, time_steps, 1)* since features, in this case, equals 1. LSTM units expect 3D input, so these dimensions fit perfectly.

5.6.3 Preprocessing of output data

Next, I need to prepare the expected results of the models in advance. The target variable in this case will be the "Close" column, which is the only column that the output data will contain. As I mentioned earlier, I need to emphasize what type of problem I will be solving. For a 1-day prediction, the output will be of shape *(samples, 1)*. While for a multi-step prediction (n days), the result will have the dimension equal to *(samples, n)*. In this case, I do not have to deal with the number of features since all models try to predict only the "Close" column. More closely, for a 1-day prediction, the output is a vector where each component represents a prediction of 1 day ahead from a certain point. In the case of the multi-day forecast, the result is a matrix that contains the prediction for n days in its rows, so if the last day we send to prediction is t, the result is $[t + 1, t + 2, ..., n]$ as shown in equation 5.2.

5.6.4 Normalization

To normalize the data, I use MinMaxScaler, from the sklearn library. In MinMaxScaler, you can define the range on which to normalize. The default value is set to $(0, 1)$. The scaler is trained on the training data using the *fit()* function. The advantage of the scaler is that it remembers the individual constants and can do the inverse operation based on these recalculations, i.e. get back the actual values.

5.7 Chapter summary

The chapter described the reasons for the choice of data used for training and validation of the models. It also mentioned where the data was collected. Afterwards, a basic description of the data and a statistical description of the target column were made. Next, it was explained how the data must be preprocessed to have the shape required for the specific type of prediction. Then the actual data splitting and normalization of the data were described.

Chapter 6

Hyperparameter tuning

The chapter describes one of many procedures for tuning hyperparameters. It is also explained why this particular procedure cannot be used in this thesis, so an alternative one is chosen. The options that could have been chosen and the subsequent final decision and its reasons are also described. Furthermore, the style of obtaining the best model with the best hyperparameters is proposed.

Hyperparameter tuning is a big part of the work in ML, as each model accepts different hyperparameters, which can take many values. Moreover, for each problem, different values of these parameters are suitable, and therefore there are many possibilities by which a given model could be trained. There is no way to determine the correct most optimal values of the hyperparameters that would fit every model.

6.1 Grid Search

Grid Search is one of the tools that can be used to find optimal hyperparameters. Grid Search uses the so-called *param_grid*, which is nothing more than the definition of hyperparameters and the values that each hyperparameter can take. In order to be able to validate the current hyperparameters in some way, Grid search uses a validation dataset. However, this dataset is taken from the training dataset using cross-validation. I will not explain cross-validation much in this thesis, but in a nutshell, it is the principle of using a window that "slides" over the entire training set. During training, this window is discarded from the data to be trained, and then the window itself is used for validation.

6.2 Grid Search and time series

Grid search variation with CV (cross-validation) uses data shuffling before cross-validation, in order to calculate the accuracy of the model more precisely. You can set this parameter to *False* so that the shuffling itself does not take place, however, Grid search still has some problems that appear for time series applications. As you may have already guessed, due to removing the window from the training set and then merging the two parts, the dependency between the training data is lost.

6.3 KerasTuner

KerasTuner is a tool that is offered by the Keras library itself and is similar to the Grid Search. It also defines what values can hyperparameters take, and then this tool optimally searches the hyperparameter space to find the best ones for the model and the current task. The difference is that the parameters are not defined in a unique object (like *params_grid* in Grid Search), but they are defined directly on the model. Thus, creating a function that returns a Keras model is necessary. Alternatively, you can create the model as a class that inherits from the *HyperModel* class, where you just override the build function. I chose the second variant since I needed to pass parameters to models.

Another difference is that Grid Search searches the whole space, i.e. it makes combinations of all hyperparameters, which in practice means that Grid Search has to go through many combinations when working with a complex model with many hyperparameters that take on a large number of different values. Moreover, if another hyperparameter is added, or another value to a hyperparameter is added, the number of combinations grows exponentially, and so does the training time.

The thing Grid Search and KerasTuner have in common is that both tools use a validation dataset for parameter validation. The validation dataset can be added to the tuner during initialization.

In my thesis, I will use KerasTuner primarily to find out:

- the optimal number of hidden layers,
- the optimal number of neurons in each layer,
- the optimal activation function,
- the value in Layer Dropout,
- other optional hyperparameters of individual networks.

6.3.1 Defining the search space

KerasTuner offers a large number of values that can be set to search for hyperparameters. These values differ in their types. Keras offers *hp.Int()* for integer types, *hp.Float()* for decimal numbers and *hp.Choice()* for choosing, for example, an activation function. As I said, these types can be set either in the function that returns the model or in a class that inherits from *HyperModel*. For example, to get the number of neurons to be in the Dense layer, the code *model.add(Dense(hp.Int('dense_layer', min_value=16, max_value=240, step=32)))* is used. You can see that Keras offers the option to specify a range on which to find the hyperparameter value. Next, the name and step size are defined. In my case, the range is from 16 to 240 neurons and step size equals 32.

6.3.2 Selection of a tuner

KerasTuner currently offers three tuners that the user can use. Namely, RandomSearch, BayesianOptimization and Hyperband.

6.3.2.1 RandomSearch

The RandomSearch tuner searches the search space randomly, as the name implies. Compared to Grid Search, it has higher performance and therefore explores the space faster. However, if the search space is too large, Random Search is not able to cover it all, and thus, the tuner may not find the optimal hyperparameters. Because of randomness, the tuner can take values of hyperparameters that may already seem incorrect at the beginning of the algorithm.

6.3.2.2 Hyperband

Compared to the RandomSearch tuner, the Hyperband tuner addresses the problem of randomness. So its search is not purely random. In the beginning, the Hyperband tuner again makes random combinations of hyperparameters, as the RandomSearch tuner did. However, the Hyperband tuner trains these given combinations on only a few epochs. It then chooses the ones that have the most promising results from these combinations, which enormously reduces the number of combinations to the next step. Next, the combinations are again trained on additional epochs. This process is repeated until the entire training process is complete or until the model hits the maximum number of epochs. At the end of this iterative algorithm, the model that has the highest accuracy is selected.

Compared to the RandomSearch tuner, the Hyperband tuner is more capable of finding more good parameter values. Since it discards those hyperparameters that are not optimal during the execution of the algorithm, it is also able to cover a more significant number of combinations of these hyperparameters at the beginning. However, the tuner still has the problem of randomly selecting hyperparameter combinations at the beginning of the algorithm.

6.3.2.3 BayesianOptimization

The last tuner offered by the Keras library is the BayesianOptimization tuner. This tuner addresses the problems of both previous tuners. Specifically, the random selection of combinations of hyperparameter values. The BayesianOptimization tuner selects only a few random combinations, from which it further determines which combinations might be better than the others. Thus, it takes into account the history of tried combinations of hyperparameter values. The algorithm runs until it finds the optimal combination of hyperparameter values or until it encounters the maximum number of trials that the user can set when initializing the tuner.

6.3.2.4 Final selection

I tested each tuner on a randomly selected model, which was the same for all tuners. Based on this comparison, I decided to use only the Hyperband tuner primarily because of the speed of the search itself.

I also send a callback function to the tuner to stop learning if the validation loss function has not changed over several epochs. This function is called *EarlyStopping()* and is provided by the Keras library. The function also helps to ensure that the model does not overfit.

6.3.3 Getting a model from KerasTuner

KerasTuner keeps the individual models that have been examined during the algorithm. In addition, Keras supports a function that can return the best n combinations of hyperparameters. In addition, there is a function that can return the n best models. This function is called *get_best_models(num_models=1)* and takes as its argument a number that expresses how many best models the user wants to return.

Since the function's output is an array, the user must query the first element if he wants to get the best model. These models are also already trained. However, the Keras documentation states that it is best to take only the best hyperparameters, rebuild the model, and train it on the whole dataset. Since I continue to send the model to a function that retrains the model and further returns the training history, I chose the option recommended by the official documentation. This way, I can get the model with the best hyperparameters.

6.4 Chapter summary

The chapter describes one of the tools that can be used for tuning hyperparameters. It also explains why this particular tool cannot be used. An alternative one is then offered in the form of KerasTuner, which satisfies what I need for the work involved. A tuner selection is then made, as the Keras library currently provides 3 of them. The reason for choosing a particular tuner is also given, followed by a description of how to obtain a model from the tuner that has the best hyperparameters and can be further trained on the data.

Chapter 7

Proposed models

This chapter describes the used models. It specifies the style in which the models are trained and how the training yielded the most optimal model. It also discusses implementation details, such as the shape of the input data. The chapter divides models into 7 groups based on the used model. Each model group has 4 variants. Furthermore, the model labelling is introduced. It also provides a description of the models and their optimal hyperparameters that were chosen in the previous chapter.

As already mentioned, the models were created using the Keras library. Specifically, I used the Sequential model, which serves the purpose of being able to add custom layers to it, and their arrangement is purely up to the user's choice. There are two ways to create a sequential model.

The first one is to define a sequential model and then add individual layers using the *add()* function. I found this method the most user-friendly, and therefore it is primarily used. However, in some cases, I had to use the second method.

The second one is defining a layer that is then added as an input to the next layer.

7.1 Model training

The individual models were extracted from KerasTuner using the method described in the previous section 6.3.3. As mentioned before, I obtained the best combinations of hyperparameters utilising this method. However, I still need to train the model on the training data. The models are sequentially trained using the *fit()* function, to which I also send a callback function as a parameter, namely *ModelCheckpoint()*, which allows me to save the model weights after each epoch. I can then load those weights with the highest accuracy on the validation set in order to get the best model.

When the model calls the training function, it returns a history showing how the model was learned and its accuracy at each epoch. An example of one of the learning histories can be seen in figure 7.1. The figure shows the training history of the Univariate LSTM 1 model proposed below.

7.2 Implementation details

The models were trained to predict from previous 15 and 10 days. In the figures mentioned below, the number of days from which the models predicted is indicated by the variable *ts*. The other divisions are the dimensions of the models' input. The split is into univariate and multivariate. The shape of the input are *(Samples, ts, 1)* for univariate and *(Samples, ts, 5)* for multivariate.

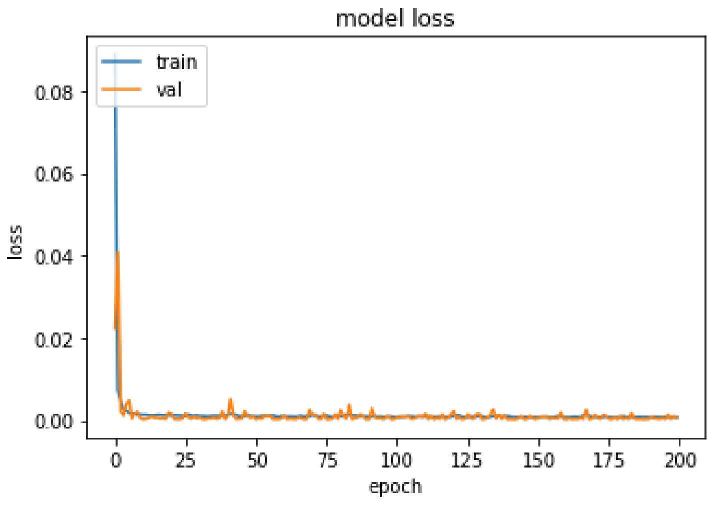


Figure 7.1 Example of Univariate LSTM 1 model learning history.

The subsequent division is how many days the model predicts. In my thesis, I consider 1-day and 5-day predictions, so the dimensions of the outputs are *(Samples, 1)* for 1-day prediction and *(Samples, 5)* for 5-day prediction.

7.3 Model labelling

Since each model can have four variants, I divide each model into a separate group, marked with a number. Then, I label each variant, specifically Univariate 1 (1-day prediction), Univariate multi-step (multi-step prediction), Multivariate 1 and Multivariate multi-step, with a letter. Thus, if I am talking about a model from group 2, the Univariate multi-step variant, I use the label 2.b.

7.4 LSTM models (1)

The first group are the LSTM models. The hyperparameters of this model include *input_unit*, which determines the number of neurons in the input LSTM layer, and then the Dropout layer with the parameter *dropout_rate*. Next is the Dense layer, which has been optionally inserted, i.e., depending on whether the network chooses to do so, the choice parameter is *add_dense_layer*. If the Dense layer has been chosen, it contains the parameter *dense_layer*, specifying the number of neurons. The last parameter *dense_activation* is the activation function of the Dense output layer. In table 7.1 you can see the values of each parameter for all 4 variants. Furthermore, in figure 7.2(a) you can see the structure of the Multivariate multi-step LSTM model with input shapes and output shapes.

7.5 Stacked LSTM models (2)

The next group is the Stacked LSTM models. The models have a parameter *input_neurons*, which specifies the number of neurons in the input LSTM layer. The models contain the parameter *n_layers* which is the number of extra hidden layers. All variants chose the number of hidden layers to be 0, so I do not consider the number of neurons in the hidden layers in the table. The next layer is again an LSTM with the number of neurons *layer_2_neurons*. Next, the models had the option to add a Dense layer, that is, the parameter *add_dense_layer*, which has the number of neurons denoted as *dense_layer*. The last parameter *dense_activation* is the activation function of the Dense output layer. The exact values of the hyperparameters can be seen in table 7.2. The structure of the Univariate Stacked LSTM 1 model can be seen in figure 7.3(a).

Parameters	1.a	1.b	1.c	1.d
input_unit	480	448	480	192
dropout_rate	0.2	0.25	0.1	0.1
add_dense_layer	True	True	True	True
dense_layer	240	208	144	176
dense_activation	linear	linear	linear	linear

Table 7.1 Table of parameters that can be set in the LSTM model. Each column describes a unique model variant proposed in section 7.3.

Parameters	2.a	2.b	2.c	2.d
input_unit	384	384	416	384
n_layers	0	0	0	0
layer_2_neurons	160	160	128	32
dropout_rate	0.1	0.25	0.25	0.15
add_dense_layer	True	True	False	True
dense_layer	112	48	-	208
dense_activation	linear	linear	linear	ReLU

Table 7.2 Table of parameters that can be set in the Stacked LSTM model. Each column describes a unique model variant proposed in section 7.3.

7.6 CNN models (3)

Another group is the CNN models. The models have again 4 variants. The hyperparameters of the first layer include *filters*, which determine the number of filters, *kernel_size*, which specifies the size of the filters. Furthermore, the network contains *pool_size*, which denotes the size of the MaxPooling operation. Another hyperparameter is *dropout_rate*, which is located in the Dropout layer. Next, the model again has a choice of an additional Dense layer based on a value of *add_dense_layer* parameter. The dense layer also has parameter *dense_layer* which specifies the number of neurons. The last hyperparameter *dense_activation* is again the activation function of the Dense output layer. Detailed hyperparameters of each variant can be seen in table 7.3. The structure of the Univariate CNN 1 model can be seen in figure 7.4(b).

7.7 CNN-LSTM models (4)

The following is a combination of CNN and LSTM models. The model has the same parameters as the CNN, namely *filters*, *kernel_size* and *pool_size*. Then, the LSTM layer follows with the parameter *LSTM_units*, which indicates the number of neurons. Next is the Dropout layer with *dropout_rate*. After that, the models again have a choice of the Dense layer using *add_dense_layer* with the number of neurons *dense_layer*. This is followed by the Dense output layer with the activation function *dense_actiovation*. All the parameters of the models can be seen in table 7.4. The structure of Univariate CNN-LSTM 1 can be seen in figure 7.3(b).

Parameters	3.a	3.b	3.c	3.d
filters	208	144	48	208
kernel_size	6	6	6	2
pool_size	2	4	2	6
dropout_rate	0.25	0.25	0.25	0.05
add_dense_layer	True	True	True	True
dense_layer	112	176	240	80
dense_activation	ReLU	ReLU	ReLU	linear

Table 7.3 Table of parameters that can be set in the CNN model. Each column describes a unique model variant proposed in section 7.3.

Parameters	4.a	4.b	4.c	4.d
filters	176	80	144	176
kernel_size	6	4	2	2
pool_size	6	2	2	6
LSTM_units	160	224	96	160
dropout_rate	0.25	0.2	0.25	0.0
add_dense_layer	False	True	False	True
dense_layer	-	144	-	80
dense_activation	linear	linear	linear	linear

Table 7.4 Table of parameters that can be set in the CNN-LSTM model. Each column describes a unique model variant proposed in section 7.3.

7.8 Bidirectional LSTM models (5)

The fifth group is the Bidirectional LSTM models. The models contain the parameter *LSTM_units*, which specifies the number of neurons in the LSTM unit that is in the Bidirectional layer. Subsequently, the models contain a Dense layer with *dense_layer* neurons. The last parameter is *activation_function*, which specifies the activation function of the Dense output layer. The detailed parameters of each variant can be seen in table 7.5. The structure of the Multivariate multi-step Bidirectional LSTM can be seen in figure 7.2(b).

Parameters	5.a	5.b	5.c	5.d
LSTM_units	448	256	128	64
dense_layer	208	208	208	240
dense_activation	linear	linear	ReLU	linear

Table 7.5 Table of parameters that can be set in the Bidirectional LSTM model. Each column describes a unique model variant proposed in section 7.3.

7.9 Stacked GRU models (6)

The next group is the Stacked GRU models. The parameters are similar to the Stacked LSTM models, except that the input GRU layer has *GRU_units* neurons. The number of hidden layers is *n_layers*. Each hidden layer has many neurons. In this case, only one extra layer was chosen, which has *gru_0_units* neurons. This is followed by another GRU layer with *gru_2_neurons* neurons. The models had the option of choosing the Dense layer using the parameter *add_dense_layer* with the number of neurons *dense_layer*. The last parameter *dense_activation* is the activation function of the Dense output layer. As I said, the Stacked GRU model is similar to the Stacked LSTM model, but LSTM is replaced by GRU. In figure 7.3(a), you can see the Univariate Stacked LSTM 1 model.

7.10 Encoder-Decoder models (7)

The last group is the Encoder-Decoder models. These models contain an encoder layer with *encoder* neurons and a decoder layer with *decoder* neurons. Next is the Dropout layer with the parameter *dropout_rate*. Like all groups, the models had the option of choosing the Dense layer using the parameter *add_dense_layer* with the number of neurons *dense_layer*. The last parameter *dense_actiovation* is the activation function of the Dense output layer. A detailed description of the hyperparameters chosen for each variation of the model can be seen in table 7.7. The structure of the Univariate Encoder-Decoder LSTM 1 model can be seen in figure 7.4(a).

Parameters	6.a	6.b	6.c	6.d
GRU_units	448	448	320	96
n_layers	0	0	0	1
gru_0_units	-	-	-	160
gru_2_neurons	32	128	96	32
dropout_rate	0.15	0.3	0.05	0.05
add_dense_layer	False	True	False	True
dense_layer	-	80	-	112
dense_activation	linear	linear	ReLU	ReLU

Table 7.6 Table of parameters that can be set in the Stacked GRU model. Each column describes a unique model variant proposed in section 7.3.

Parameters	7.a	7.b	7.c	7.d
encoder	448	160	448	224
decoder	512	384	192	416
dropout_rate	0.3	0.15	0.2	0.0
add_dense_layer	True	True	False	False
dense_layer	448	512	-	-
dense_activation	ReLU	ReLU	sigmoid	linear

Table 7.7 Table of parameters that can be set in the Encoder-Decoder LSTM model. Each column describes a unique model variant proposed in section 7.3.

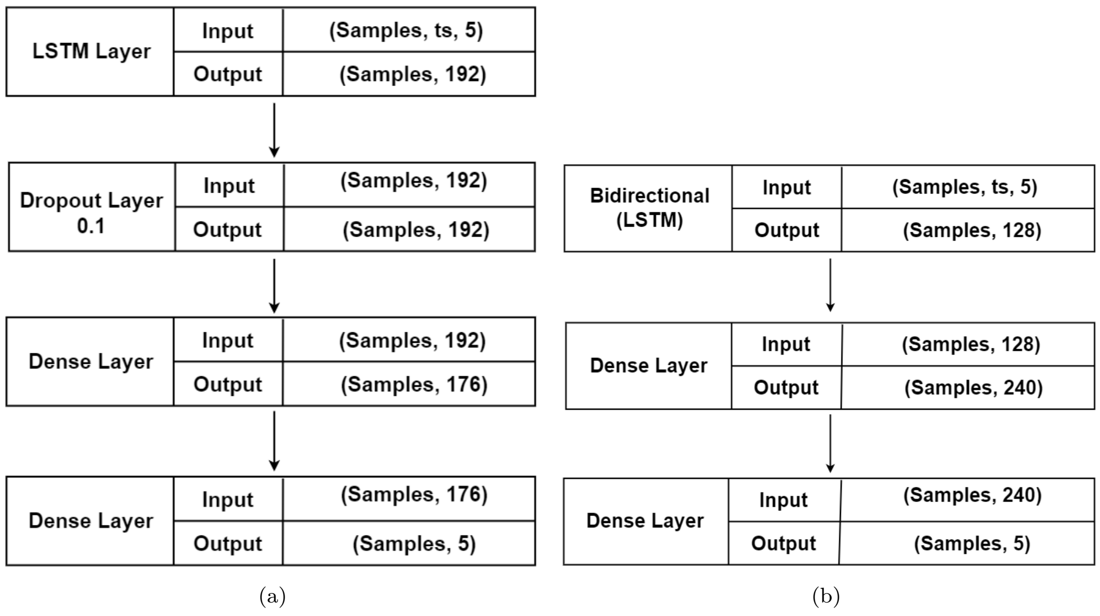


Figure 7.2 (a) Structure of the Multivariate multi-step LSTM model (1.d) with output and input shapes of each layer. (b) Structure of the Multivariate Bidirectional multi-step LSTM model (6.a) with output and input shapes of each layer.

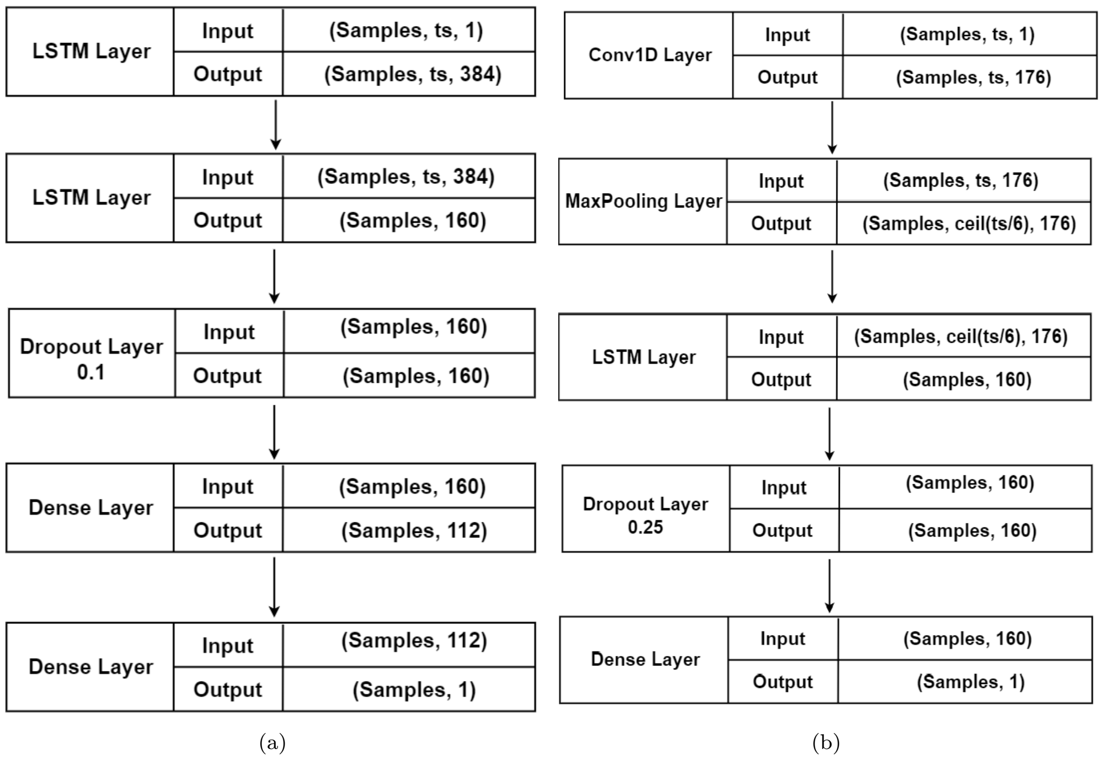


Figure 7.3 (a) Structure of the Univariate Stacked LSTM 1 model (2.a) with output and input shapes of each layer. (b) Structure of the Univariate CNN-LSTM 1 model (4.a) with output and input shapes of each layer.

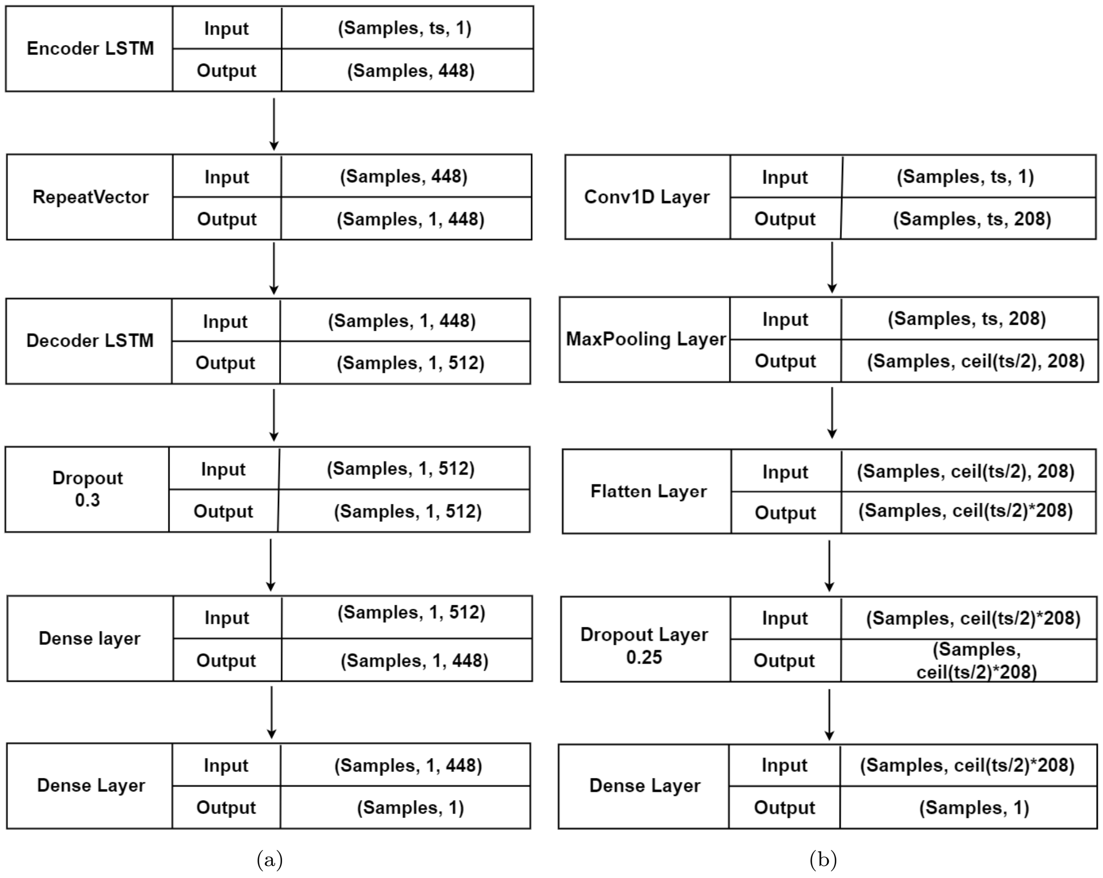


Figure 7.4 (a) Structure of the Univariate Encoder-Decoder LSTM 1 model (7.a) with output and input shapes of each layer. (b) Structure of the Univariate CNN 1 model (3.a) with output and input shapes of each layer.

7.11 Chapter summary

The chapter introduced the implemented models. First, it was presented how the models were created, how they were trained, and how the weights with the best accuracy were obtained from the training process. Next, the implementation details were presented, precisely the shape of the input data, which is divided into two groups: univariate and multivariate. Then the shape of the output for 1-day prediction and 5-day prediction is presented. Next, the models themselves were divided into groups, each containing 4 model variants. The way in which I labelled each model variant was presented. The models and their variants were then presented along with their hyperparameters selected in the previous chapter. At the end of the chapter, the structures of the selected models with the shapes of the inputs and outputs of each layer were shown.

Chapter 8

Experiments and results

In this part of the thesis, the individual experiments are described. The experiments are divided into parts according to the shape of the input and the shape of the output. The particular model validations are shown in the tables presented in the chapter.

8.1 Experiments

Experiments were conducted for 1-day and multi-step prediction (5-day). The final evaluation was based on the RMSE value. For 1-day prediction, RMSE is calculated from the predicted day. For multi-step prediction, RMSE is calculated on every day of the forecast. Thus, if the model predicts five days, the RMSE will be calculated for all five days. Then the resulting overall RMSE is calculated, which the models tried to minimize. Thus, experiments can be divided into predictions of 1 or more days and further based on the shape of the model input: multivariate or univariate. The other experiment that I have done is the effect of the number of days that the model uses for prediction. In the thesis, 10 and 15 days were tested. The last experiment I conducted was to retrain the selected models on a larger dataset. The dataset consists of the previous 5 years of stock data. This dataset is preprocessed the same as the last one. I would also like to mention that a lot depends on the actual training of the model. If the model is retrained, it can have better or worse results.

8.1.1 1-day prediction

8.1.1.1 10 days

As can be seen in table 8.1, for the 1-day prediction, the best performing model is called Univariate LSTM 1 with RMSE equal to 3.807, and then the Univariate Stacked GRU 1 with 3.81 RMSE.

8.1.1.2 15 days

For the option of 15 days from which the model should predict the next day, the model called Univariate CNN-LSTM 1 with an RMSE value of 3.956 works best. Another model with similar results is the Univariate LSTM 1, with an RMSE value of 4.109.

Model	Dataset	10 days	15 days
Univariate LSTM 1 (1.a)	Train	3.708	3.431
	Val	2.368	2.411
	Test	3.807	4.109
Multivariate LSTM 1 (1.c)	Train	4.257	4.146
	Val	3.052	3.408
	Test	6.253	6.610
Univariate Stacked LSTM 1 (2.a)	Train	3.557	3.530
	Val	2.444	2.661
	Test	4.441	4.302
Multivariate Stacked LSTM 1 (2.d)	Train	3.911	3.558
	Val	3.251	3.217
	Test	10.075	9.575
Univariate CNN 1 (3.a)	Train	3.502	3.148
	Val	2.298	2.433
	Test	4.106	4.260
Multivariate CNN 1 (3.c)	Train	3.744	3.518
	Val	5.338	5.224
	Test	13.245	13.658
Univariate CNN-LSTM 1 (4.a)	Train	3.429	3.318
	Val	2.495	2.453
	Test	4.084	3.956
Multivariate CNN-LSTM 1 (4.c)	Train	5.426	4.722
	Val	2.908	3.310
	Test	8.747	11.926
Univariate Bidirectional LSTM 1 (5.a)	Train	3.367	3.353
	Val	2.406	2.508
	Test	4.030	4.124
Multivariate Bidirectional LSTM 1 (5.c)	Train	3.317	100.341
	Val	3.707	173.512
	Test	9.286	207.356
Univariate Stacked GRU 1 (6.a)	Train	3.556	3.359
	Val	2.392	2.372
	Test	3.810	4.119
Multivariate Stacked GRU 1 (6.c)	Train	4.664	4.881
	Val	2.735	3.467
	Test	5.480	8.332
Univariate Encoder-Decoder LSTM 1 (7.a)	Train	3.350	3.356
	Val	2.377	2.507
	Test	3.885	4.145
Multivariate Encoder-Decoder LSTM 1 (7.c)	Train	4.972	4.261
	Val	3.090	3.158
	Test	6.059	8.034

Table 8.1 Prediction of 1 day from the previous 10 and 15 days. The first column describes the model's name. The second column represents which dataset was used and the last two columns describe the RMSE value of 10 used days and 15 used days.

8.1.1.3 Overall

In general, it can be seen that for almost all models, the ones based on 10 days, from which they have to predict the next one, work better compared to 15 days. Only for the Univariate CNN-LSTM 1 and the Univariate Stacked LSTM 1 models, 15 days have better results than 10 days. Detailed results of all models can be seen in table 8.1.

8.1.2 5-day prediction

8.1.2.1 10 days

Another experiment is the prediction of 5 days from the previous 10 days. For this problem, the best model is 7.b, i.e. the Univariate multi-step Encoder-Decoder LSTM model with RMSE value equal to 5.882 overall 5 days. Another model with similar results is the model with an overall RMSE value equal to 5.979. This model 6.b is called Univariate multi-step Stacked GRU. The detailed outcomes of the models can be seen in Table 8.2.

8.1.2.2 15 days

The models are then evaluated based on the previous 15 days for the 5-day prediction. The best model in this aspect was the 4.b model, i.e. the Univariate multi-step CNN-LSTM model with an RMSE value of 6.038. The second best model with a similar value is model 6.b, i.e. the Univariate multi-step Stacked GRU model, with an RMSE value of 6.103. The detailed results of the models based on 15 days for predicting the next 5 days can be seen in Table 8.3.

8.1.2.3 Overall

Again, based on tables 8.2 and 8.3, it can be seen that for the prediction of 5 days, the models that predict from the previous 10 days work better than those that predict from 15 days. For almost all models, the 10-day variant performs better. However, some models perform better for 15 days.

8.2 Chapter summary

The chapter introduced the individual experiments. These experiments are divided according to the shape of the input. The next division is by how many days the models used for forecasting. In the thesis, I considered 10 and 15 days. Another division is by the number of columns the models used for prediction. The last division is based on the number of days the models predict. Experiments were conducted for the prediction of 1 and 5 days. The chapter contains the performance of each model, shown in tables.

Model	Dataset	t+1	t+2	t+3	t+4	t+5	overall
1.b	Train	5.179	5.632	6.482	6.462	6.636	6.105
	Val	2.658	3.004	3.504	3.851	4.132	3.472
	Test	3.492	4.796	5.375	7.660	8.871	6.347
1.d	Train	4.220	5.081	5.955	5.873	6.122	5.496
	Val	3.151	3.893	34.683	4.249	4.591	4.151
	Test	5.757	6.240	6.436	8.507	9.749	7.496
2.b	Train	3.719	4.655	5.089	5.570	5.980	5.063
	Val	2.562	3.258	3.870	4.114	4.411	3.702
	Test	3.931	4.807	5.766	6.895	8.223	6.115
2.d	Train	3.857	4.429	4.971	5.378	5.769	4.928
	Val	2.890	3.527	3.877	4.138	4.456	3.816
	Test	7.060	8.338	8.789	9.950	9.851	8.862
3.b	Train	3.962	4.621	5.598	5.723	6.118	5.264
	Val	2.316	3.000	3.607	3.809	4.089	3.423
	Test	4.029	5.551	5.231	7.092	8.712	6.334
3.d	Train	3.593	4.781	5.923	5.593	5.730	5.196
	Val	3.988	4.591	5.513	4.184	4.946	4.676
	Test	10.429	10.377	7.583	8.073	11.091	9.614
4.b	Train	4.035	4.794	5.370	5.816	6.381	5.341
	Val	2.931	3.273	3.592	4.089	4.167	3.641
	Test	5.005	5.622	6.072	7.857	8.513	6.749
4.d	Train	5.109	5.412	6.339	6.559	6.521	6.019
	Val	2.907	3.613	3.589	3.834	4.566	3.740
	Test	8.267	10.314	9.094	8.968	9.392	9.231
5.b	Train	3.710	4.488	5.134	5.558	5.922	5.025
	Val	2.786	3.474	3.898	4.280	4.686	3.881
	Test	3.957	4.839	5.655	6.966	8.411	6.170
5.d	Train	4.090	4.971	5.544	5.742	6.020	5.318
	Val	3.440	4.288	4.493	4.168	4.733	4.247
	Test	6.451	6.687	6.985	7.969	8.993	7.477
6.b	Train	3.798	4.699	5.417	5.763	6.188	5.241
	Val	2.443	3.164	3.530	4.019	4.063	3.496
	Test	3.712	4.450	5.536	7.233	7.887	5.979
6.d	Train	3.989	4.677	5.281	5.787	5.961	5.190
	Val	3.587	4.101	4.352	4.640	4.732	4.302
	Test	6.931	7.492	7.513	8.628	10.019	8.191
7.b	Train	3.289	4.198	4.943	5.568	5.943	4.882
	Val	2.287	3.129	3.550	3.953	4.247	3.501
	Test	3.631	5.350	5.341	6.463	7.801	5.882
7.d	Train	4.291	5.031	5.574	5.997	6.320	5.490
	Val	3.532	3.683	3.875	4.073	4.355	3.914
	Test	5.444	7.876	7.427	8.599	9.499	7.887

Table 8.2 Prediction of 5 days from the previous 10 days. The first column describes the label of a model. The next column represents the dataset that was used. The following six columns represent the RMSE value of each day. Thus, the 3rd column is the RMSE value of the first day. The 4th column represents the RMSE value of the second predicted day, and so on. The last column defines the RMSE value through all predicted days.

Model	Dataset	t+1	t+2	t+3	t+4	t+5	overall
1.b	Train	3.474	4.309	4.919	5.375	5.681	4.817
	Val	2.422	3.139	3.630	3.856	4.067	3.473
	Test	3.900	4.842	6.010	7.151	8.480	6.290
1.d	Train	4.054	4.657	5.184	5.536	5.840	5.094
	Val	3.150	4.081	3.848	4.040	4.229	3.888
	Test	6.227	8.040	6.736	8.268	9.279	7.788
2.b	Train	3.537	4.367	5.048	5.453	5.764	4.899
	Val	2.548	3.255	3.709	4.002	4.218	3.596
	Test	4.158	5.008	5.533	7.176	8.458	6.261
2.d	Train	4.581	5.093	5.574	5.848	6.149	5.477
	Val	3.607	3.999	4.409	4.529	5.067	4.350
	Test	6.371	7.449	7.964	9.690	10.052	8.419
3.b	Train	3.474	4.262	4.755	5.201	5.380	4.666
	Val	2.335	3.100	3.740	4.085	4.412	3.611
	Test	4.325	5.286	6.879	7.547	8.841	6.769
3.d	Train	3.789	4.403	4.963	5.252	5.531	4.828
	Val	4.716	5.675	6.282	6.561	6.522	5.992
	Test	10.345	12.746	10.837	14.760	16.154	13.159
4.b	Train	3.360	4.198	4.835	5.305	5.611	4.731
	Val	2.417	3.118	3.678	3.915	4.192	3.521
	Test	3.799	4.735	5.843	6.875	8.002	6.038
4.d	Train	5.172	5.706	6.194	5.786	5.973	5.776
	Val	3.537	4.441	4.001	4.461	5.139	4.349
	Test	7.518	7.219	11.697	9.491	13.248	10.110
5.b	Train	3.422	4.214	4.844	5.242	5.452	4.693
	Val	2.530	3.330	3.930	4.238	4.636	3.805
	Test	3.976	4.742	6.095	7.252	8.947	6.451
5.d	Train	4.506	5.371	5.705	5.818	7.041	5.747
	Val	3.740	4.490	4.598	4.890	5.873	4.768
	Test	8.470	11.356	12.320	14.428	10.822	11.642
6.b	Train	3.625	4.574	5.211	5.651	5.850	5.047
	Val	2.647	3.127	3.530	3.770	3.958	3.439
	Test	4.546	4.727	5.542	6.749	8.184	6.103
6.d	Train	4.912	5.472	5.735	6.060	6.346	5.726
	Val	2.967	3.368	3.672	3.893	4.157	3.635
	Test	5.798	7.907	9.522	10.566	10.057	8.940
7.b	Train	3.565	4.310	4.833	5.231	5.491	4.736
	Val	2.718	2.958	3.459	3.724	3.931	3.389
	Test	3.665	6.304	5.932	7.142	8.361	6.469
7.d	Train	3.908	4.640	5.164	5.528	5.823	5.058
	Val	3.379	3.543	3.794	3.981	4.222	3.796
	Test	5.369	8.126	7.248	8.513	9.496	7.874

Table 8.3 Prediction of 5 days from the previous 15 days. The first column describes the label of a model. The next column represents the dataset that was used. The following six columns represent the RMSE value of each day. Thus, the 3rd column is the RMSE value of the first day. The 4th column represents the RMSE value of the second predicted day, and so on. The last column defines the RMSE value through all predicted days.

Chapter 9
Discussion

In this chapter, a discussion of the individual experiments is presented. It also points out the flaws of the repeatedly used 1-day prediction, which can be found in most scientific papers. This method is compared with the multi-step prediction. Subsequently, the best model is tested to predict the specific 5 days of the test dataset. Then a discussion about the experiment with a larger dataset is presented.

9.1 Iterative 1-day vs multi-step prediction

In this thesis, I try to highlight the problem of many articles that address the topic of stock market prediction. Most of the papers try to predict only 1 day, which is not applicable in practice for many reasons.

The first reason is that the predicted day is further used for the next prediction. Since no model is error-free, the error produced by the prediction of the first day is reflected in the next day's forecast since the model uses this incorrectly predicted day to predict the next one. The consequence is that the resulting error increases exponentially.

Another problem is that the models mentioned in the papers learn to predict almost the same number they got on the last day of the input. In practice, the papers show figures that are similar to picture 9.4. The graph is excellent, and many people will say that the model predicts beautifully, but if the figure is zoomed in, its shortcomings become apparent. As you can see in image 9.5, the entire prediction is shifted by 1 day to the right. As mentioned, this is because the model has learned to return a very similar number to what it was on the last day of input data. The biggest problem with this approach is that if the model were to predict 5 days iteratively, the resulting prediction would just become a straight line, as you can see in figure 9.1. This makes this type of prediction useless in practice. A better approach is multi-step prediction, which you can see in figure 9.3. The trained models try to predict 5 days ahead and not just 1.

I want to mention that only the selected univariate model was used in the thesis for iterative prediction since all my models predict only close prices. The multivariate model could not use this predicted value and thus could not predict iteratively.

9.2 Multi-step prediction

Figure 9.2 shows the multi-step prediction for the test data. This is specifically a prediction of 5 days ahead of each point. As you can see, the model tried to capture the pattern in the data, but the data are so different from each other that it failed to do so. It can predict some days quite well, but it could have happened by chance.

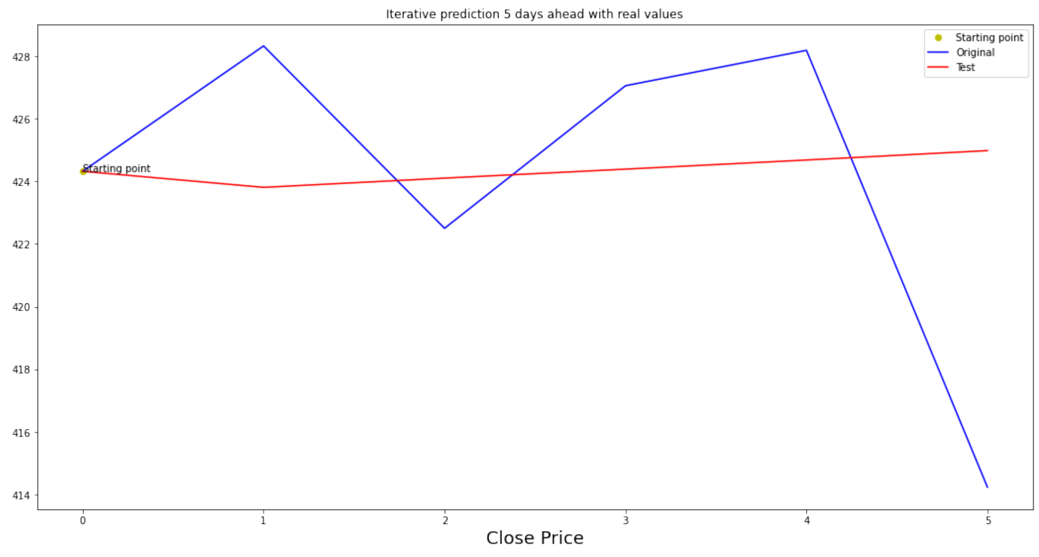


Figure 9.1 Example of an iterative 1-day prediction. As you can see, the model is unusable. The 5 days mentioned are days from the penultimate week of test data.

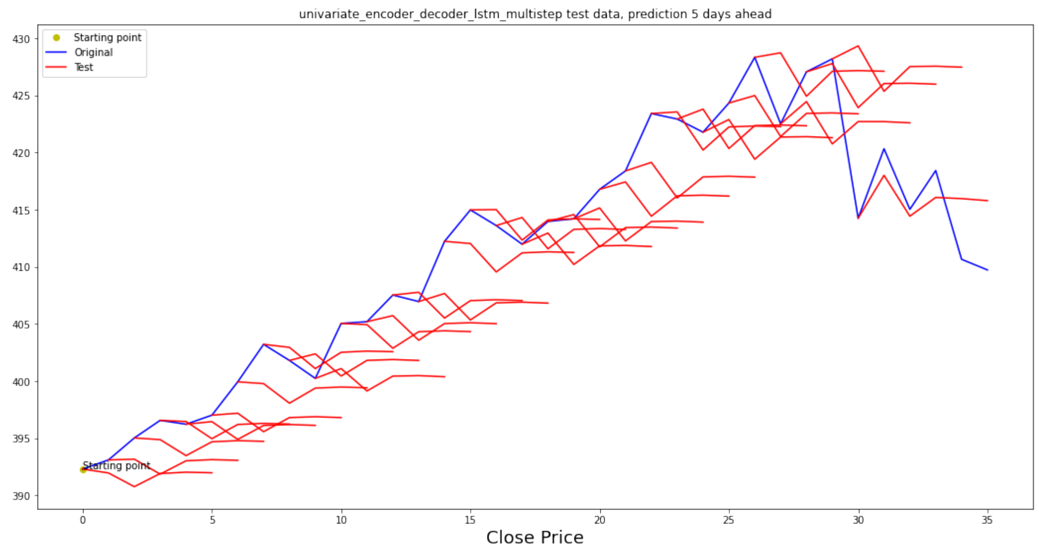


Figure 9.2 Graph showing the 5-day prediction of the best multi-step model. Specifically, this is the test data set.

9.3 Last 5 days prediction

For the selected models, the prediction was tested on the last 5 days of the test dataset. The results can be seen in figure 9.3. However, this is only for the selected 5 days. For other choices of data, the results might be completely different. As you can see, the model tried to capture a specific pattern. However, it is essential to note that it is highly dependent on the choice of data and the actual training of the model.

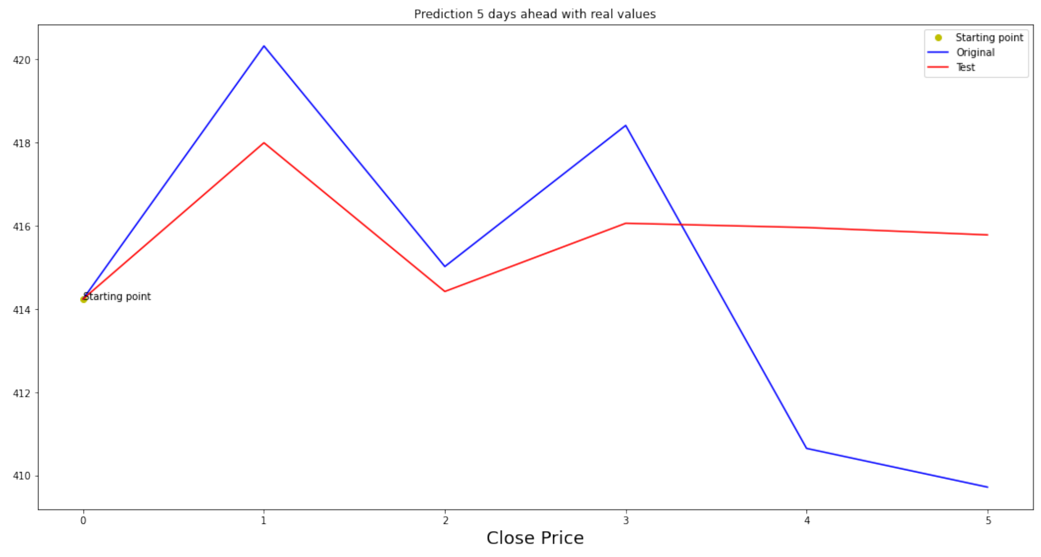


Figure 9.3 Example of the best multi-step model. The model is evaluated on last 5 days of testing data.

9.4 Larger dataset

The selected models that were most successful for both the 1-day and 5 days predictions were used and retrained on larger data to test if the prediction success rate would increase. This success rate did not improve, and for example, for the best model (7.b) for the 5-day prediction, the overall RMSE increased to 7.725. For the 1-day prediction, the resulting difference was not as significant. The RMSE increased to 4.754 for model 1.a, which was the best one-step model.

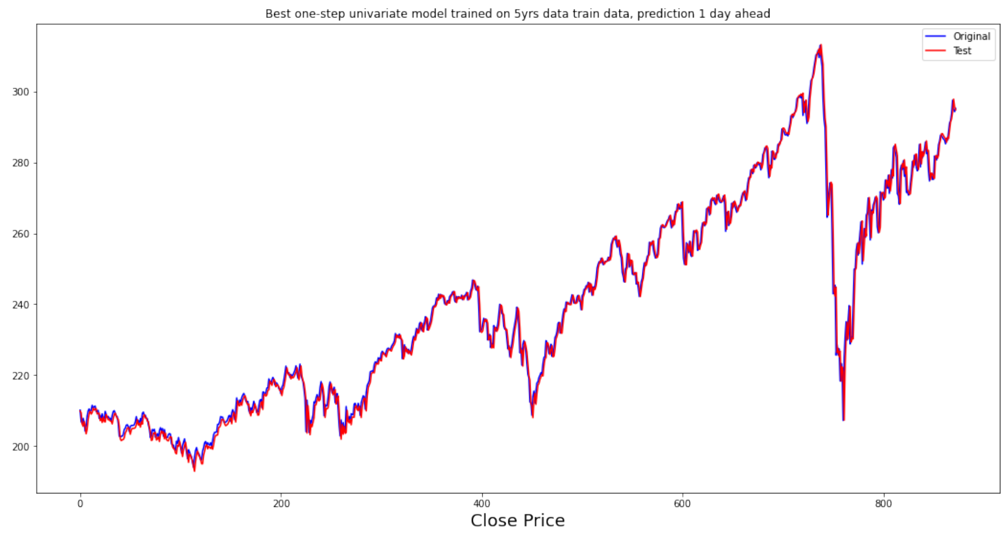


Figure 9.4 Example of the best model that predicts 1-day prediction. The model is trained on data collected over a period of 5 years. The graph shows the training data.

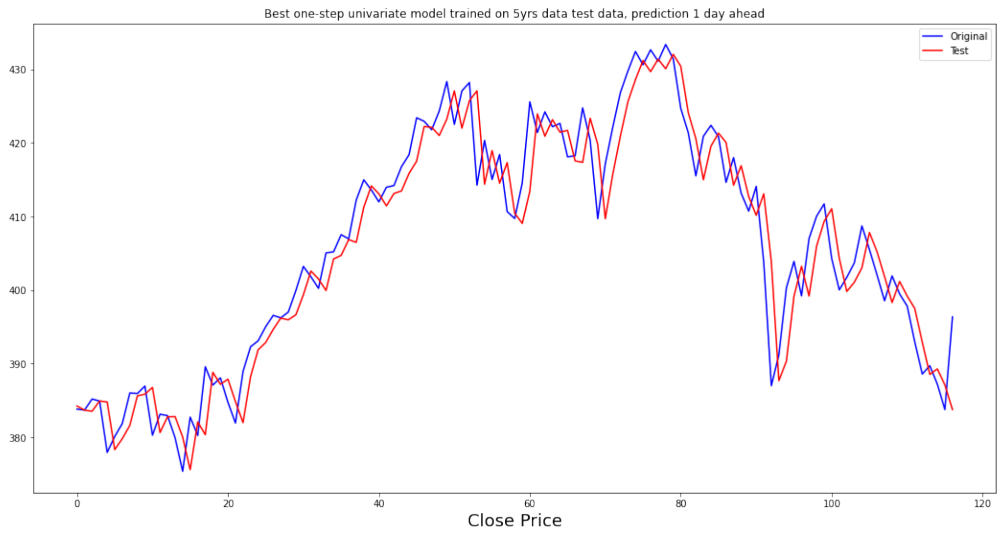


Figure 9.5 Example of the best model for 1-day prediction. The model is evaluated on test data of a data collected over a period of 5 years.

9.5 Selection of best model

Since the 1-day prediction is useless in practice, as I showed above, I can eliminate all models that predict only 1 day. This leaves only multi-step models, for which the choice of data and also the learning itself is very important, as the RMSE can vary by as much as 0.5. For my particular training and my particular data, the Encoder-Decoder multi-step LSTM model works best, specifically its univariate variant, i.e. model 7.b. The model is tested to predict 5 selected days of test dataset, which can be seen in figure 9.3.

9.6 Future work

Many improvements are on the table for the future. The primary change from the current solution is to use world news or posts on social networks that relate to a given stock market in a prediction. Subsequently, obtaining the sentiment of these news headlines will determine the news's emotion. People react based on these emotions, so it is pivotal to include the given feature in the work. Extracting the sentiment of the news itself can be done by using the LSTM model that has been introduced in the thesis. There are other possibilities here, and it is essential to explore the issue further.

9.7 Chapter summary

The chapter highlighted the problem of repeatedly using a 1-day prediction. It was explained why this method is flawed and was compared with a multi-step approach that had better results. Furthermore, the selected model for multi-step prediction was tested on the last 5 days of the test dataset. The same was done for the iterative 1-day prediction where the mentioned problems could be shown. At the end of the chapter, the best multi-step model and the best 1-day model were tested on a larger dataset containing data collected over 5 years. Also, the future work that can be done was mentioned. Also, adding a sentiment of news headlines was noted, which can help to improve the models.

Chapter 10

Conclusion

The main aim of this work was to create a model with the highest accuracy in predicting selected stock market. Firstly, a literature overview of existing solutions was carried out. The research is divided into several parts. The first part is the traditional statistical approach. The exploration shows that this approach is not as successful as the ML approach. The survey showed that the most commonly used models in ML are CNN, LSTM and GRU. However, even this approach is not the best and has its shortcomings mentioned in this thesis. Next, papers that take into account both historical data and news from the world that affect the development of stock prices are presented. The last approach is fundamental analysis, which aims to analyse the company as a whole.

The most famous models were explained and implemented in the next chapters. Each model has 4 variations based on the input and output shape. These variations are Univariate, Univariate multi-step, Multivariate and Multivariate multi-step. The optimal hyperparameters for each model were tuned using the KerasTuner tool. From the tuner, the untrained models with the best hyperparameters were created. The models were further trained on selected training data.

The data on which the models were trained and validated were selected from Yahoo Finance website, which provides individual stock markets and their historical data. The stock market that the models attempted to predict is the ETF SXR8.DE, which replicates the S&P 500 index, which is one of the most famous indexes in the world. The basic description and statistical description of the data were provided. The data had to be split and preprocessed. Since the input and output shape differs for each problem, the preprocessing part had to be done separately for each of them.

Later, the models were evaluated for both 1-day and 5-day predictions of the selected stock market. In addition, several different approaches have been suggested as to how each model is evaluated. The results show that the best model is the Univariate LSTM 1 model for 1-day prediction and the Univariate multi-step Encoder-Decoder LSTM model for 5-day forecasting. The model Univariate LSTM 1 had an RMSE equal to 3.807 on testing data. Similarly, the Univariate multi-step Encoder-Decoder LSTM model had an overall RMSE equal to 5.882 on testing data. Based on the results from the experiments, the best model was selected and then tested on a larger dataset.

In the thesis, I also warn about the iterative reuse of single-day prediction in order to predict multiple days. Many papers show prediction charts and claim that the model predicts well. However, the experiments proposed in this work show that this approach is unusable as the predictions are biased by the overfitting of the models.

The main goal of the thesis and all the sub-goals were successfully fulfilled. There are still many improvements that can be applied in order to increase the model accuracy. For example, adding sentiment of news headlines or tweets related to the selected stock market, as these things have a significant influence on the price development. How the sentiment of news headlines can be obtained is described in section 9.6. Since I am interested in this topic, I have decided to work on the topic as part of the VýLeT (Výzkumné léto na FIT) programme, where I would like to integrate news and tweets sentiments into the selected models.

Bibliography

1. ARIYO, Adebiyi A.; ADEWUMI, Adewumi O.; AYO, Charles K. Stock Price Prediction Using the ARIMA Model. In: *2014 UKSim-AMSS 16th International Conference on Computer Modelling and Simulation*. 2014, pp. 106–112. Available from DOI: `10.1109/UKSim.2014.67`.
2. SAMAL, K. Krishna Rani; BABU, Korra Sathya; DAS, Santosh Kumar; ACHARAYA, Abhirup. Time Series Based Air Pollution Forecasting Using SARIMA and Prophet Model. In: *Proceedings of the 2019 International Conference on Information Technology and Computer Communications*. Singapore, Singapore: Association for Computing Machinery, 2019, pp. 80–85. ITCC 2019. ISBN 9781450372282. Available from DOI: `10.1145/3355402.3355417`.
3. EAPEN, Jithin; BEIN, Doina; VERMA, Abhishek. Novel Deep Learning Model with CNN and Bi-Directional LSTM for Improved Stock Market Index Prediction. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 2019, pp. 0264–0270. Available from DOI: `10.1109/CCWC.2019.8666592`.
4. MEESAD, Phayung; RASEL, Risul Islam. Predicting stock market price using support vector regression. In: *2013 International Conference on Informatics, Electronics and Vision (ICIEV)*. 2013, pp. 1–6. Available from DOI: `10.1109/ICIEV.2013.6572570`.
5. HOSEINZADE, Ehsan; HARATIZADEH, Saman. CNNpred: CNN-based stock market prediction using a diverse set of variables. *Expert Systems with Applications*. 2019, vol. 129, pp. 273–285. ISSN 0957-4174. Available from DOI: `https://doi.org/10.1016/j.eswa.2019.03.029`.
6. GUPTA, Ishu; MADAN, Tarun Kumar; SINGH, Sukhman; SINGH, Ashutosh Kumar. *HiSA-SMFM: Historical and Sentiment Analysis based Stock Market Forecasting Model*. arXiv, 2022. Available from DOI: `10.48550/ARXIV.2203.08143`.
7. AASI, Bipin; IMTIAZ, Syeda Aniqa; QADEER, Hamzah Arif; SINGARAJAH, Magdalean; KASHEF, Rasha. Stock Price Prediction Using a Multivariate Multistep LSTM: A Sentiment and Public Engagement Analysis Model. In: *2021 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*. 2021, pp. 1–8. Available from DOI: `10.1109/IEMTRONICS52119.2021.9422526`.
8. KHAN, Wasiat; GHAZANFAR, Mustansar Ali; AZAM, Muhammad Awais; KARAMI, Amin; ALYOUBI, Khaled H.; ALFAKEEH, Ahmed S. Stock market prediction using machine learning classifiers and social media, news. *Journal of Ambient Intelligence and Humanized Computing*. 2020. ISSN 1868-5145. Available from DOI: `10.1007/s12652-020-01839-w`.

9. SELVIN, Sreelekshmy; VINAYAKUMAR, R; GOPALAKRISHNAN, E. A; MENON, Vijay Krishna; SOMAN, K. P. Stock price prediction using LSTM, RNN and CNN-sliding window model. In: *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. 2017, pp. 1643–1647. Available from DOI: `10.1109/ICACCI.2017.8126078`.
10. APPLICATION OF ARTIFICIAL NEURAL NETWORKS IN HYDROLOGY, ASCE Task Committee on. Artificial neural networks in hydrology. I: Preliminary concepts. *Journal of Hydrologic Engineering*. 2000, vol. 5, no. 2, pp. 115–123.
11. LE, Xuan Hien; HO, Hung; LEE, Giha; JUNG, Sungho. Application of Long Short-Term Memory (LSTM) Neural Network for Flood Forecasting. *Water*. 2019, vol. 11, p. 1387. Available from DOI: `10.3390/w11071387`.
12. VAŠATA, Daniel; KLOUDA, Karel. *VZD Přednášky*. 2022. Available also from: `https://kam.fit.cvut.cz/bi-vzd/lectures/index.html`.
13. NIELSEN, Michael A. *Neural networks and deep learning*. Determination press San Francisco, CA, USA, 2015.
14. OLAH, Christopher. *Understanding LSTM Networks*. 2015. Available also from: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.
15. LIPTON, Zachary C.; BERKOWITZ, John; ELKAN, Charles. *A Critical Review of Recurrent Neural Networks for Sequence Learning*. arXiv, 2015. Available from DOI: `10.48550/ARXIV.1506.00019`.
16. CHEN, Gang. *A Gentle Tutorial of Recurrent Neural Network with Error Backpropagation*. arXiv, 2016. Available from DOI: `10.48550/ARXIV.1610.02583`.
17. WERBOS, P.J. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*. 1990, vol. 78, no. 10, pp. 1550–1560. Available from DOI: `10.1109/5.58337`.
18. PASCANU, Razvan; MIKOLOV, Tomas; BENGIO, Yoshua. On the difficulty of training recurrent neural networks. In: *International conference on machine learning*. 2013, pp. 1310–1318.
19. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long Short-term Memory. *Neural computation*. 1997, vol. 9, pp. 1735–80. Available from DOI: `10.1162/neco.1997.9.8.1735`.
20. GERS, Felix A.; SCHMIDHUBER, Jürgen; CUMMINS, Fred. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*. 2000, vol. 12, no. 10, pp. 2451–2471. Available from DOI: `10.1162/089976600300015015`.
21. GERS, Felix; SCHMIDHUBER, Jürgen; CUMMINS, Fred. Learning to Forget: Continual Prediction with LSTM. *Neural computation*. 2000, vol. 12, pp. 2451–71. Available from DOI: `10.1162/089976600300015015`.
22. STAUDEMEYER, Ralf C; MORRIS, Eric Rothstein. Understanding LSTM–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*. 2019.
23. SHERSTINSKY, Alex. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network. *Physica D: Nonlinear Phenomena*. 2020, vol. 404, p. 132306. ISSN 0167-2789. Available from DOI: `https://doi.org/10.1016/j.physd.2019.132306`.
24. GERS, Felix; SCHMIDHUBER, Jurgen. Recurrent nets that time and count. In: 2000, vol. 3, 189–194 vol.3. ISBN 0-7695-0619-4. Available from DOI: `10.1109/IJCNN.2000.861302`.
25. CHEN, Jing; HUANG, Xinyu; JIANG, Hao; MIAO, Xiren. Low-Cost and Device-Free Human Activity Recognition Based on Hierarchical Learning Model. *Sensors*. 2021, vol. 21, p. 2359. Available from DOI: `10.3390/s21072359`.
26. CHO, Kyunghyun; MERRIENBOER, Bart van; BAHDANAU, Dzmitry; BENGIO, Yoshua. *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*. arXiv, 2014. Available from DOI: `10.48550/ARXIV.1409.1259`.

27. CHUNG, Junyoung; GULCEHRE, Caglar; CHO, KyungHyun; BENGIO, Yoshua. *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling.* arXiv, 2014. Available from DOI: 10.48550/ARXIV.1412.3555.

28. CUI, Zhiyong; KE, Ruimin; PU, Ziyuan; WANG, Yinhai. *Deep Bidirectional and Unidirectional LSTM Recurrent Neural Network for Network-wide Traffic Speed Prediction.* arXiv, 2018. Available from DOI: 10.48550/ARXIV.1801.02143.

29. ALTHELAYA, Khaled A.; EL-ALFY, El-Sayed M.; MOHAMMED, Salahadin. Stock Market Forecast Using Multivariate Analysis with Bidirectional and Stacked (LSTM, GRU). In: *2018 21st Saudi Computer Society National Computer Conference (NCC).* 2018, pp. 1–7. Available from DOI: 10.1109/NCG.2018.8593076.

30. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep learning.* MIT press, 2016.

31. GRAVES, A.; SCHMIDHUBER, J. Framewise phoneme classification with bidirectional LSTM networks. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* 2005, vol. 4, 2047–2052 vol. 4. Available from DOI: 10.1109/IJCNN.2005.1556215.

32. SCHUSTER, Mike; PALIWAL, Kuldip. Bidirectional recurrent neural networks. *Signal Processing, IEEE Transactions on.* 1997, vol. 45, pp. 2673–2681. Available from DOI: 10.1109/78.650093.

33. ARISOY, Ebru; SETHY, Abhinav; RAMABHADRAN, Bhuvana; CHEN, Stanley. Bidirectional recurrent neural network language models for automatic speech recognition. In: *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP).* 2015, pp. 5421–5425. Available from DOI: 10.1109/ICASSP.2015.7179007.

34. CHO, Kyunghyun; MERRIENBOER, Bart van; GULCEHRE, Caglar; BAHDANAU, Dzmitry; BOUGARES, Fethi; SCHWENK, Holger; BENGIO, Yoshua. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation.* arXiv, 2014. Available from DOI: 10.48550/ARXIV.1406.1078.

35. MEHTAB, Sidra; SEN, Jaydip; DUTTA, Abhishek. Stock Price Prediction Using Machine Learning and LSTM-Based Deep Learning Models. In: THAMPI, Sabu M.; PIRAMUTHU, Selwyn; LI, Kuan-Ching; BERRETTI, Stefano; WOZNIAK, Michal; SINGH, Dhananjay (eds.). *Machine Learning and Metaheuristics Algorithms, and Applications.* Singapore: Springer Singapore, 2021, pp. 88–106. ISBN 978-981-16-0419-5.

36. LECUN, Yann; BENGIO, Y. Convolutional Networks for Images, Speech, and Time-Series. In: 1995.

37. HOSEINZADE, Ehsan; HARATIZADEH, Saman. *CNNPred: CNN-based stock market prediction using several data sources.* arXiv, 2018. Available from DOI: 10.48550/ARXIV.1810.08923.

38. GAMBOA, John Cristian Borges. Deep learning for time-series analysis. *arXiv preprint arXiv:1701.01887.* 2017.

39. YAMASHITA, Rikiya; NISHIO, Mizuho; DO, Richard Kinh Gian; TOGASHI, Kaori. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging.* 2018, vol. 9, no. 4, pp. 611–629. ISSN 1869-4101. Available from DOI: 10.1007/s13244-018-0639-9.

40. O'SHEA, Keiron; NASH, Ryan. *An Introduction to Convolutional Neural Networks.* arXiv, 2015. Available from DOI: 10.48550/ARXIV.1511.08458.

41. LIVIERIS, Ioannis E.; PINTELAS, Emmanuel; PINTELAS, Panagiotis. A CNN–LSTM model for gold price time-series forecasting. *Neural Computing and Applications.* 2020, vol. 32, no. 23, pp. 17351–17360. ISSN 1433-3058. Available from DOI: 10.1007/s00521-020-04867-x.

Contents of attached medium

```
README.md ........................... a brief description of medium structure
thesis ............ directory with source codes of the text of the bachelor thesis
    text ............................................... directory with text
    ctufit-thesis.tex ........................... source of the thesis in LaTeX
    thesis.pdf ....................................... thesis in PDF form
src ................................... source codes of the implementation
    README.md ...................... a brief description of the implementation
    requirements.txt ............ required libraries used in the implementation
    stock_prediction.ipynb ................ source code of the implementation
    datasets .............................. datasets used in the implementation
```