UNIVERSITA' DEGLI STUDI DI PADOVA

DIPARTIMENTO DI SCIENZE ECONOMICHE ED AZIENDALI "M.FANNO"

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA MAGISTRALE IN ECONOMICS AND FINANCE

TESI DI LAUREA

DEEP LEARNING WITH LONG SHORT-TERM MEMORY FOR STOCK MARKET PREDICTIONS AND PORTFOLIO OPTIMIZATION

RELATORE:

CH.MO PROF. CLAUDIO FONTANA

LAUREANDO: FELICE FIORENTINO

MATRICOLA N. 1207030

ANNO ACCADEMICO 2020 – 2021

Il candidato dichiara che il presente lavoro è originale e non è già stato sottoposto, in tutto o in parte, per il conseguimento di un titolo accademico in altre Università italiane o straniere.
Il candidato dichiara altresì che tutti i materiali utilizzati durante la preparazione dell'elaborato sono stati indicati nel testo e nella sezione "Riferimenti bibliografici" e che le eventuali citazioni testuali sono individuabili attraverso l'esplicito richiamo alla pubblicazione originale.

The candidate declares that the present work is original and has not already been submitted, totally or in part, for the purposes of attaining an academic degree in other Italian or foreign universities. The candidate also declares that all the materials used during the preparation of the thesis have been explicitly indicated in the text and in the section "Bibliographical references" and that any textual citations can be identified through an explicit reference to the original publication.

Firma dello studente

Felice Fiorentino

ACKNOWLEDGEMENT

I would like to express my deep and sincere gratitude to my supervisor, Professor Claudio Fontana, who helped me to develop this work with his advices, comments, and suggestions. His recommendations inspired me and help to investigates a lot of aspects of this research to improve clarity. It was a great privilege to work under his guidance and I was grateful for the opportunity offered.

I am extending my heartful thanks to Matteo Rizzi and Ilaria Poletto for the work review, and to my family for their love and sacrifices for educating and preparing me for the future.

ABSTRACT

Long short-term memory (LSTM) network is a machine learning model which can be used for financial series forecasting. We set up LSTM networks for predicting stock prices for the constituent stocks of the S&P 500 from 2005 until 2019. We create various LSTM architecture to understand the impact on price forecasting. After the selection of the best architecture in terms of predictions, we investigate the performance across the various constituents of S&P 500 index. The model presents a varying performance in relation to the constituent: in average the stock movement accuracy is 57%. We build a portfolio daily rebalanced based on these predictions, for the test set of our dataset represented by the last three years (from 2017 to 2019). Despite the accuracy of the model is around 70% for 82 stocks, the portfolios constructed do not present a good performance. The Sharpe ratio in annual terms after transaction costs are: 1.19 in 2017, -0.57 in 2018 and -0.50 in 2019.

Contents

Introduction 4

1 Stock price forecasting and quantitative trading 5

1.1 Quantitative trading 6
1.2 Momentum trading 7
1.3 Quantitative trading: a new approach 8
1.3.1 Adjusted closing price of the stock 9
1.4 Portfolio evaluation and cumulative return 10
1.5 Value at Risk and Conditional Value at Risk 11
1.6 Sharpe ratio 13

2 LSTM algorithm for stock price prediction 14

2.1 Basics of machine learning 14
2.2 Deep learning algorithm 15
2.3 Activaction functions 18
2.3.1 Sigmoid function 18
2.3.2 Hyperbolic tangent function 18
2.4 Model optimization 19
2.5 Back-propagation 22
2.6 Overfitting 25
2.7 Recurrent neural network: architecture and optimization problem 27
2.8 LSTM algorithm 30
2.9 Stock return prediction model 32
2.10 Hardware 33
2.11 Hyperparameters selection 33
2.12 Feature scaling 34
2.13 The optimizer: ADAM 37
2.14 Regularization strategies: dropout and early-stopping 40

3 Model results and investment strategy 43

3.1 Model set up 43
3.2 Algorithm performance across constituents 49
3.3 Accuracy of predictions 50
3.4 Investment strategy 52
3.5 Portfolio LSTM prediction based and S&P500 index: a comparison 57
3.6 Model drawbacks 60

Conclusion **64**

Introduction

The stock price prediction represents a notoriously difficult task in finance considering the many elements that impact on future price movements. Identifying the stock price evolution can help investors avoid risks and obtain higher returns; for these reasons, this task has become a hot field and attracted many researchers' attention.

There are many techniques and statistical models proposed in academic literature to predict the stock price given its past observed values. In recent years, the computerization of the markets encourages automation and algorithmic solutions. In particular, the success of machine learning has attracted the financial communities interest to obtain new models for various tasks, including stock price predictions. Machine learning is a branch of Artificial Intelligence (AI) based on a specific approach: learn from data, identify patterns and make decisions with minimal human intervention. In recent years, Artificial Intelligence and machine learning innovations have had a strong impact in different fields, from the manufacturing industry to social media services. Furthermore, there are promising finance applications: process automation, fraud detection, portfolio management and algorithmic trading. Risk assessment represents another interesting area of application for deep learning models. Examples of risk assessment problem are: bankruptcy prediction, credit scoring and evaluation, mortgage evaluation (Ozbayoglu et al. 2020, p. 14). Machine learning algorithms try to make predictions to solve these tasks thanks to the big dataset.

In the last years, initial evidence has been established that machine learning techniques are capable of identifying (non-linear) structures in financial market data as explained in various works, for instance: Dixon et al. (2015), Krauss et al. (2016), Hao & Gao (2020) and Fischer & Krauss (2017). The aim of this work is to investigate the effectiveness of the machine learning approach for stock price predictions. Specifically, we want to apply a specific model called Long short-term memory: one of the most advanced machine learning architectures for sequence learning tasks, such as handwriting recognition, speech recognition, or time series prediction (Graves 2013). Our task is to predict the daily stock prices for the constituents of S&P500 index, which includes 500 large companies listed on U.S. stock exchanges. From these predictions, we can calculate the stock return to build a portfolio of stocks which contains the best stocks in term of daily returns (long position) and the worse stocks (short position). The portfolio is rebalanced each day following the stock price predictions.

We proposed a different LSTM architecture compared to the similar works in literature. Our goal is try to overcome some drawbacks and extend this model to a different periods than the studies cited before. We proposed various LSTM architectures to understand how the

results change. Furthermore, we investigated the model performance across the various constituents. This is the principal contribution of this work, there is a analysis on how the performance change across the various S&P500 constituents and the relative impact on portfolio construction.

This work is organized as follows. The first chapter presents stock price prediction in literature and the definition of quantitative trading, an investment strategies based on mathematical models and algorithms. The machine learning approaches are related to these kinds of trading systems. Then we illustrate our investment strategies to build a portfolio and the relative metrics to measures its financial performance.

The second chapter illustrates the machine learning approach and the LSTM algorithm. After a brief overview of the basics concepts, we illustrate the various aspects of our algorithm. We provide a deep explanation of the mathematics behind this model. We focus our attention on all the hyperparameters that can influence the model performance and the various strategies to overcome overfitting problem to improve the performance.

The third chapter presents the hyperparameters selected for our model and the underlying reasons that justify this choice. Then we present the result of our algorithm for the daily stock price prediction and the various problems that we have encountered. Finally, we present our daily portfolio strategy based on these predictions to understand this approach's effectiveness. These portfolios are evaluated considering the of daily returns, the relative risk metrics, and cumulative return performance with a yearly horizon.

Chapter 1

Stock price forecasting and quantitative trading

The stock (or equity) is a security that represents ownership of a fraction of a company. It gives the right to receive part of cash flows of a company, in form of dividend payments. Hence, the stock prices can be decomposed as:

$$\text{stock price} = \text{NPV of future expected dividend} + \text{resale value}$$

The value and payments claims (future dividends) can rise and fall according to the financial markets and the company's fortunes. The stock price change, knowing as the return, represents a source of profit for the investors if it is positive. There are various models for stock prediction. For example, there are statistical approaches which try to identify the stock price considering its historical observed values, but the stock price predictions with past data is challenging, given the high degree of noise present in the market. Furthermore, we have to consider an important theory presented by Fama (1965) and Fama (1970): the efficient market hypothesis. There are three degrees of market efficiency:

- **Weak form:** All available information are incorporated in the stock price. Hence, past price movements are not useful for predicting future prices.
- **Semi-strong form:** Stock prices adjust quickly to absorb new public information. The investor cannot benefit from new information and achieves superior returns.
- **Strong form:** The stock prices reflect all the public and private information. In this case, not even a corporate insider can achieve superiors returns.

In an efficient market, there is no possibility to predict potential prices from historical data or from other types of information. The stocks are trading at their fair value. It is impossible for investors to either buy undervalued stocks or sell overvalued stocks. Hence, they cannot beat the market.

Yet, there is a plethora of well-known capital market anomalies that are in stark contrast with the notion of market efficiency (Fischer & Krauss 2017). For example, Jacob (2015) and Green et al. (2013) provide surveys that collect more than 100 of such capital market anomalies, which effectively rely on return predictive signals to outperform the market.

Given this possibility, various trading strategies have been proposed to rely on market anomalies. In particular, we want to analyze the profitability of a quantitative trading strategy, introduced in recent years with the automatization of exchange markets and the Information Technology (IT) revolution.

The following section presents the quantitative trading innovations and our approach to build a strategy to gain profits based on machine learning. The technical details of this approach will be present in Chapter 2.

1.1 Quantitative trading

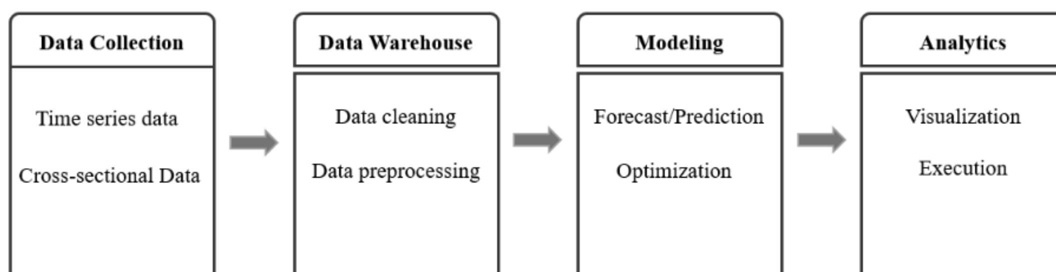
Starting with a general definition of quantitative trading.

Definition 1.1. *Quantitative trading consists of trading strategies based on quantitative investment analysis, which relies on mathematical models to design an automated trading system. Market trend, entry and exit trade, price history and volume are the critical factors for each quantitative trading strategy (Ta et al. 2020).*

Quantitative trading works by using data-based models to determine the probability of a certain outcome happening. The basic idea is to build an architecture to leverage statistics, computer algorithms, and computational resources for high-frequency trading systems, aiming to minimize risk and maximize return based on the historical performance of the encoded strategies tested against historical data.

The process involves various steps: after collecting and analyzing historical data, we can construct an algorithm to make accurate predictions on price or returns and build a specific investment strategy. Specifically, the algorithm tries to identify recurrent patterns in stock price and their relative probability to make predictions. This model must be optimized through back-testing. Finally, the results of the analysis are visualized and become the criteria for investment decision-making.

Figure 1.1: Quantitative investment management system



Source: Ta et al. (2020)

Quantitative trading popularity increased in the last decade, is generally used by financial institutions and hedge funds. It represents the new era of trading, which provided different benefits: lower commissions, anonymity, control, and analysis on a big set of stock which is

difficult for a human being, discipline (weeds out the emotion of fear and greed and promotes rational decisions), access, competition and reduced transaction costs (Ta et al. 2020).

However, these quantitative models are profitable only for a particular market type or condition for which the model was made. They need to be redeveloped as the market conditions evolve. The deep learning algorithm is specifically applied for the stock price prediction and selection based on the historical dataset.

Quantitative trading involves short term strategy and a substantial Information Technology structure (IT). The stock price prediction represents the starting point of various strategies which can be executed with quantitative trading.

Our investment strategy is based on selecting a certain number of stocks which must be hold over a short horizon of time. The stocks selection is inspired from momentum strategy, presented in the next section.

1.2 Momentum trading

In academic literature, momentum trading is a famous strategy based on the predictability of stock returns. Momentum trading is the practice of buying and selling assets according to the recent strength of price trends. It is based on the idea that if there is enough force behind a price move, it will continue to move in the same direction.

An important article that demonstrates this strategy is Jegadeesh & Titman (1993). The authors demonstrated that stocks which performed well historically continued to perform well over a subsequent certain interval of time. In particular, they select stocks based on the relative returns over the past J months and holds them for subsequent K months (also called J-month/K-month strategy). The analysis was conducted on New York stock exchange stocks from 1965 to 1989. J and K are set equal to various period: 3,6,9 and 12 months. At the beginning of each month, securities are ranked in ascending order based on their returns in the past J months. Based on these rankings, ten decile portfolios are formed that equally weight the stocks contained in the top decile, the second decile, and so on. They construct the portfolios in such a way that the winner portfolio consisted of the best performing stocks, and the loser portfolio consisted of the worst performing stocks. This strategy leads to a return of 12.01% per year on average.

The profitability of this strategy, according to the authors, is due to the fact that individuals tend to overreact to information. Daniel et al. (1998) argue that investors will buy a stock if good news supports their optimistic indication. On the other hand, investors do not sell the stock if the negative indication is inconsistent with positive news or their previous opinion. Jegadeesh & Titman (1993) also indicate that the market under-reacts to information about the short-term prospects of firms but over-reacts to information about their long-term prospects.

The conservatism bias, identified in experiments by Edwards (1968), suggests that individuals underweight new information in updating their priors. If investors act in this way, prices will tend to slowly adjust to information, but once the information is fully incorporated in prices there is no further predictability in stock returns. Barbens, Shleifer & Vishny (1998) propose that under-reaction is initiated by the conservatism bias over time horizons of one to fourteen months.

However, the momentum strategy illustrated above is based on a long-term horizon of investment. Quantitative trading, introduced in section 1.1, usually works with short-term investment. For instance, the portfolios constructed with quantitative momentum trading are daily rebalanced. This means that we buy winners or sell losers stocks with daily frequency. This strategy's benefits are that we can obtain excess daily returns, resulting in large profits for investors. We can capitalize on the large volatility with this strategy. Suppose that a certain stock is characterized by big daily price variation, this can results in more profits for investors because the price variation can be high. However, there is also the possibility that this results in big losses.

Other problems of short-term strategy is that an investor has a huge stock turnover, which is related to fees and commissions that must be paid. Furthermore, the stock price has a lot of variation during the day, and it is necessary to understand the perfect timing when it is optimal to buy or sell a stock.

1.3 Quantitative trading: a new approach

We proposed a strategy based on forecasting daily price of various constituents of S&P500: the stock market index, which measures the performance of 500 large companies stock listened on U.S. stock exchange. The predictions are based on a new approach: deep neural network, a subset of machine learning. The machine learning represents a subset of the artificial intelligence, which consists in a model that is able to learn the parameters automatically with limited human intervention.
The theory and all the aspects of the machine learning will be presented in Chapter 2.

We want to predict a daily prices of stock considering its historical pries observation: time series forecasting. The succession of values in a time series is usually influenced by some external (or exogenous) information. If this information is not known, only the past values of the series itself can be used to build a model (Lendasse et al. 2000). We can define the stock price (P) prediction with a mathematical function of the form:

$$P_{t+1} = f_\theta(P_t, P_{t-1}, P_{t-2}, \cdots, P_{t-N+1}) \tag{1.1}$$

Where the unknown value P_{t+1} is estimated from the current and past values of P. The parameters of the model f are chosen according to the information available, i.e. to all

known values of P; this step is called learning or fitting. There are various models for time series predictions, based on linear or nonlinear functions (see Hamilton (1994)).

In the last years, initial evidence has been established that machine learning techniques are capable of identifying (non-linear) structures in financial market data, as explained in Dixon et al. (2015), Krauss et al. (2016), and Fischer & Krauss (2017).

After the daily stock price predictions and the calculation of the relative return, we select the best and worse performing stock in term of daily returns following Ta et al. 2020 and Fischer & Krauss 2017. We decided to estimate the stock price on daily basis for the characteristic our deep learning approach, which required a considerable amount of data, and because this is consistent with the quantitative trading strategy (short term horizon of investment). This approach is similar to the momentum model proposed by Jegadeesh & Titman (1993), but the portfolio is daily rebalanced in this case. The various elements of this strategy will be explained in chapter 3.

1.3.1 Adjusted closing price of the stock

The purpose of our non-linear model is the price forecasting. We collected the value of historical price series of a particular stock to set our model, but it must be considered that there are various kind of prices. Specifically, we can have:

- **Open price:** It is the price at which stock first trades upon the opening of an exchange on a trading day.
- **closing price:** It refers to the last price at which a stock trades during a regular trading session, for example in many U.S. market the regular sessions run from 9:30 a.m. to 4:00 p.m. Eastern Time.
- **Adjusted closing Price:** It refers to the 'Closing price' adjusted by accounting any corporate action. In general, it is used to analyze the historical return.

Regarding the last kind of price, we can have different operations that require adjustments, for example:

- **Stock split:** The company issues more share to current shareholder without changing the market capitalization. This leading to a stock price reduction.
- **Right offering:** Entitles the shareholders to subscribe to the rights issue in proportion to their shares. This operation will lower the value of existing shares because supply increases, hence it have a diluting effect on the existing shares.
- **Dividend:** Announcement and distribution of cash and stock dividends affect the stock price.

We used the adjusted closing price as input data of our daily stock price forecasting model to avoid the effect of the adjustments presented above. In this way the historical prices series can be evaluated without consider the company announcements on dividends. Once we obtained the results of this forecasting model, the daily prices of stock, we calculated the daily variations of price forecasting to obtain the relative returns. Finally, we constructed our portfolio based on a certain number stocks, the selection strategy will be explained in Chapter 3 section 3.4. The last step is to evaluate these portfolios daily performance.

1.4 Portfolio evaluation and cumulative return

From our strategy of stock selection, we obtained the daily returns which can be positive (gains) or negative (losses). In chapter 3, we evaluate the portfolios daily return distribution for a certain horizon of time (e.g. one year) with the classical statistics metrics as mean (μ) and median. An important measures are variance σ^2 and standard deviation (volatility) σ. These measures indicates the dispersion of daily returns of selected portfolio, high values indicate that the distribution of return has a certain degree of uncertainty.

Other metrics used are the Skewness and Kurtosis, which can analyze the portfolio daily return and understanding the impact of eventually outliers values. The Skewness represents the third moment and is a standard measure of symmetry of a distribution. Given a random variable X:

$$skew[X] = \frac{\mathbb{E}[(X - \mathbb{E}[X])^3]}{\sigma^3} \tag{1.2}$$

If the bulk of the data is at the left and the right tail is longer, the distribution is right skewed or positively skewed; if the peak is toward the right and the left tail is longer, the distribution is left skewed or negatively skewed.

Kurtosis represents the fourth moment and provides a measure of the relative weight of the tails with respect to the central body of a distribution.

$$kurt[X] = \frac{\mathbb{E}[(X - \mathbb{E}[X])^4]}{\sigma^4} \tag{1.3}$$

- A normal distribution has kurtosis exactly 3. Any distribution with kurtosis $\approx$3 (excess $\approx$0) is called mesokurtic.
- If kurtosis < 3, the distribution is called platykurtic. Compared to a normal distribution, its tails are shorter and thinner, and often its central peak is lower and broader.
- If kurtosis > 3 is called leptokurtic. Compared to a normal distribution, its tails are longer and fatter, and often its central peak is higher and sharper.

Furthermore, we can evaluate the gains of our investments strategy in a horizon of time different from one day (month or yearly). Hence, we extend the definition of the simple return to the cumulative return, which is the aggregate amount that the investment has gained or lost

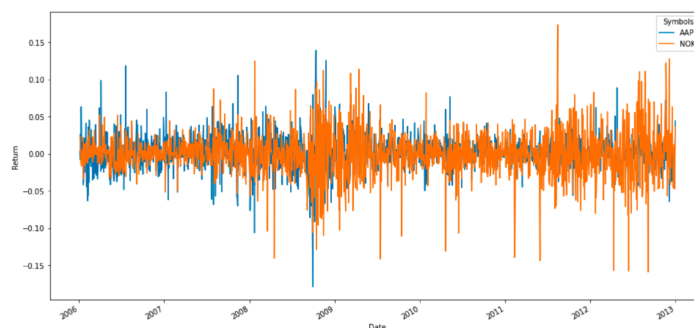
over a specific horizon of time:

$$r_{t \to T} = \left[\prod_{i=t+1}^{T} (1 + r_i) - 1 \right] \tag{1.4}$$

Where r_i is the simple return. The cumulative returns provide a clear idea of an asset's (or portfolio) performance across time. We use the adjusted closing prices, which already incorporate the effect of dividends and other corporate operations explained before. Hence, we do not consider these adjustments when evaluating the cumulative return performance. Figure 1.2 illustrates a comparison between the simple return and cumulative return, based on adjusted closing price, of Apple and Nokia's stocks. The data are collected from Yahoo finance website, for a reference period from January 2006 until December 2012. When we look at the simple return chart, the Nokia returns are more volatile than Apple, but it is not easy to understand the stock pattern across time and make a fair comparison. Instead, the cumulative return chart makes this comparison clearer; indeed, we can see how the two stocks followed different patterns after 2008.

Figure 1.2: Simple and Cumulative return comparison

(a) Apple and Nokia simple returns



(b) Apple and Nokia cumulative returns

Source: Data from Yahoo Finance

1.5 Value at Risk and Conditional Value at Risk

The daily returns obtained from our investment strategy are evaluated with two risk metrics: the Value-at-risk and Conditional-value-at-risk. Given the frequency distribution of daily

returns for a certain horizon of time (e.g 6 months or one year), we can estimate the risk.

Let us define these two metrics in general terms. Considering a X a random variable with cumulative distribution function $F_X(z)$. The random variable X in our case represents the daily return, and may have a meaning of loss or gain. In our case, X represents the loss. (Sarykalin et al. 2008, p. 272).

Definition 1.2. *The VaR of X with the confidence level* $\alpha \in]0,1[$ *is*

$$VaR_\alpha(X) = \min\{z | F_X(z) \geq \alpha\} \tag{1.5}$$

By definition, $VaR_\alpha(X)$ is a lower α-percentile of random variable X. The Value-at-risk is used in different fields involving uncertainty. The VaR is the quantile that represents the worst scenario of a certain distribution. If we consider the distribution of daily returns over a specified period of time, the VaR measures the amount of potential loss, or in other words the negative returns, that could happen which a certain probability. For example, the 5% VaR represents the 5% of total cumulative distribution associated with the worse scenario.

An interesting example of Value-at-risk application regards financial regulation, which are represented by the three Basel agreement (Basel I, Basel II, Basel III). These are agreements formed with the purpose of creating an international regulatory framework for managing credit risk and market risk. Their key function is to ensure that banks hold enough cash reserves to meet their financial obligations and absorb unexpected loss derived from financial and economic distress. The Basel Committee published 'The 1996 Amendment' which indicates that the capital that a bank must hold on its trading book is calculated as k times the VaR measured (Hull 2006, p. 495). The regulator chooses the level of k in relation to the various banks; Basel II and III picked up this measure.

Another measure of risk is called Conditional-Value-at-Risk or CVaR, which was introduced by Rockafellar & Uryasev (2000). The word 'Conditional' is due to the fact that for random variable with continuous distribution function, $CVaR_\alpha(X)$ equals the conditional expectation of X subject to $X \geq VaR_\alpha(X)$.

Definition 1.3. *The CVaR of X with confidence interval* $\alpha \in]0,1[$ *is the mean of the generalized* α*-tail distribution:*

$$CVaR_\alpha(X) = \int_{-\infty}^{\infty} z dF_X^\alpha(z) \tag{1.6}$$

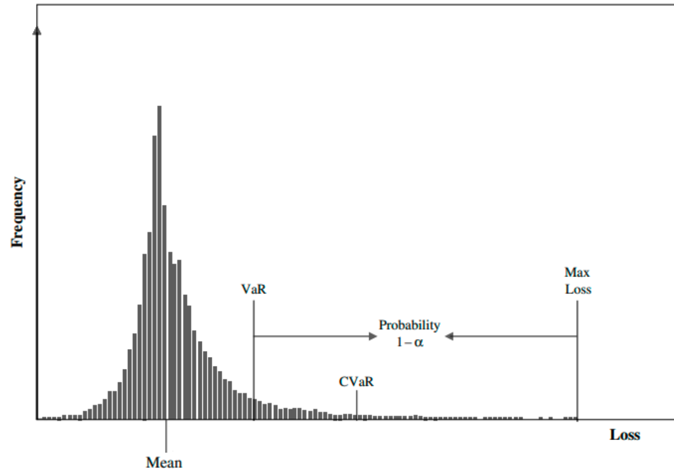
where:

$$F_X^\alpha(z) = \begin{cases} 0, & \textit{when } z < VaR_\alpha(X) \\ \frac{F_x(z)-\alpha}{1-\alpha}, & \textit{when } z \geq VaR_\alpha(X) \end{cases} \tag{1.7}$$

The CVaR is also called Expected Shortfall. While VaR is a quantile of return distribution associated with worst-case on a specific horizon of time, CVaR is the expected loss if that

worst-case threshold is ever crossed. Hence, the definition of Expected Shortfall is derived from quantifying the expected losses beyond the VaR breakpoint. Figure 1.3 represents these two percentile risk measures in a general case.

Figure 1.3: Risk functions: graphical representation of VaR and CVaR



Source:Sarykalin et al. (2008)

1.6 Sharpe ratio

The Sharpe ratio is an index introduce by William Sharpe is frequently used to evaluate the performance of an asset and portfolio. Given a certain portfolio $w \in \mathbb{R}^N$, the correspond Sharpe ratio is defined as follow:

$$SR(w) = \frac{\tilde{r}_w - r_f}{\sigma(\tilde{r}_w)} \tag{1.8}$$

$\tilde{r}_w$ represents the expected return of the portfolio and $\sigma(\tilde{r}_w)$ is the associated volatility. r_f indicates the risk-free rate of the asset the return on an investment with zero risk. The yield for a U.S. Treasury bond, for example, could be used as the risk-free rate.

This equation represents the ratio between the return risk premium of a portfolio and the standard deviation of the associated random return. We can use the Sharpe ratio to evaluate a portfolio's past performance (ex-post), by using the past returns and historical volatility in the formula. However, the investors can also calculate the ex-ante ratio using the expected portfolio return and expected risk-free return.

Chapter 2

LSTM algorithm for stock price prediction

Artificial Intelligence represents the branch of computer science dealing with the simulation of intelligent behavior in computers to think like a human. Machine learning is a particular subset of the Artificial Intelligence world regarding building machines that can learn. The first part of this chapter provides the basics of the machine learning approach. The second part, from section 2.8, illustrates our daily stock price prediction algorithm: the Long term short-memory.

2.1 Basics of machine learning

A famous definition of machine learning is provided by Mitchell (1997):

Definition 2.1. *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."*

Hence we can identify three principal elements of the machine learning algorithms. The first component is the task (T), which is usually described in terms of how the machine learning system should processes an example, which is a collection of features that have been quantitatively measured from some objects or events. The process of 'learning' refers to the ability to perform the task (Goodfellow et al. 2016, p. 99). Many kinds of task can be solved with machine learning, some of the most common are:

- **Classification**: In this type of task, the computer program is asked to specify which of k categories some input belongs to. An example of a classification task is object recognition, where the input is an image (usually described as a set of pixel brightness values), and the output is a numeric code identifying the object in the image.

- **Regression**: The computer program tries to predict a numerical value given some input. An example of a regression task is the prediction of the expected claim amount that an insured person will make (used to set insurance premiums) or the prediction of future prices of securities.

- **Structured output**: Structured output tasks involve any task where the output is a vector (or other data structure containing multiple values) with important relationships

between the different elements. An example is pixel-wise segmentation of images, where the computer program assigns every pixel in an image to a specific category.

As the second element, we have the performance measure (P) that evaluates a machine learning algorithm abilities. It is necessary to understand the goodness of the model. There are various kinds of performance measures, but the choice depends on the algorithm task type. For example, considering the classification task, we often measure the model's accuracy to evaluate P. The accuracy represents the proportion of examples for which the model produces the correct output. Nevertheless, we can also obtain equivalent information by measuring the error rate: the proportion of examples for which the model produces an incorrect output (Goodfellow et al. 2016, p. 103).

The algorithm is allowed to experience (E) on a certain dataset: a collection of examples, also called data points or collection of data. The dataset is divided into two subsets: the first is the training set (in-sample data) composed of the observations which served as inputs in order to train the model (process of learning); the second subset is called the test set or out-of-sample data and indicates the data used to provide an unbiased evaluation of a final model fit on the training dataset. In other words, the test set is used to "test" the machine learning program, obtained after the training phase. In relation to these three elements, we have different algorithm approaches.

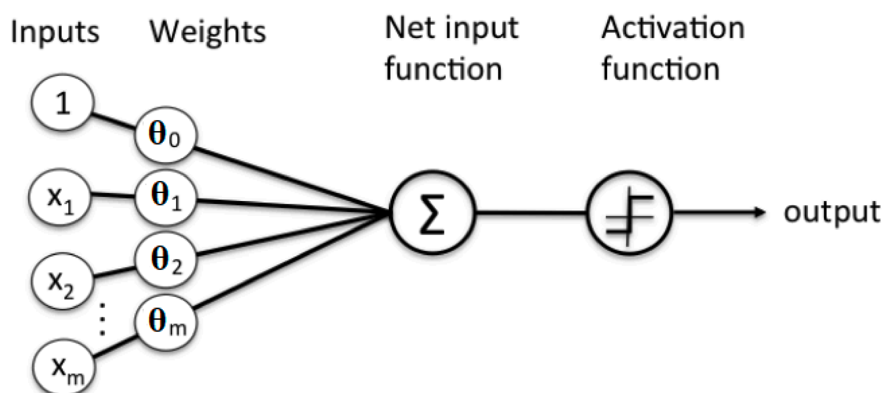
2.2 Deep learning algorithm

The machine learning algorithm world offers different architectures which can be implemented for various purposes. This work investigates the application of a method called Deep Learning. An example of this approach is provided by Dixon et al. (2015), which represents a pioneering application for financial series prediction.

The starting point to understand Deep learning, is a crude electronic model based on the brain's neural structure and neurons' role: the neural network. The biological neurons take input from numerous other neurons through the dendrites, perform the required processing on the input and send another electrical pulse through the axon into the terminal nodes from where it is transmitted to numerous other neurons. We can switch this process in an electronic model called perceptron, illustrated in Figure 2.1.

The model computes its output through a non-linear function, called activation function, over the inputs dataset. Its output depends on the input x and weights θ, representing the influence of various observations of the dataset on output. The activation function indicates how the inputs are processed to obtain the output desired by the task. The weights have to be learned. However various relationships between input and output cannot be model using a single perceptron. The quintessential of deep learning is a network of different neurons (or perceptron) linked each other (like in our brain): the multilayer-perceptron (or feedforward network).

Figure 2.1: Perceptron structure



Source: Singh et al. (2017)

The goal of this network is to approximate a specific function f^* which identifies the relation between the input x and output y. The feedforward network function is given by:

$$y = f(x; \theta) \tag{2.1}$$

The model must "learn" the value of the parameters θ, the weights of perceptron, to obtain the best function approximation. The word "feedforward" derives from the fact that information flows through the function being evaluated from x, through the intermediate computations used to define f, and finally to the output y. There are no feedback connections in which outputs of the model are feedback into itself. When feedforward neural networks are extended to include feedback connections, they are called recurrent neural networks.

These models are called "network" because they are typically represented by composing many different functions (Goodfellow et al. 2016, p. 168). For example, we might have three different functions $f^{(1)}$, $f^{(2)}$ and $f^{(3)}$ connected in a chain to form:

$$f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})) \tag{2.2}$$

Each function represents a layer of the network, also know as hidden layer, and the last one is called output layer. Any layer is composed by different neurons (also know as perceptron). During neural network training, we drive $f(\mathbf{x})$ to match $f^*(\mathbf{x})$. The training data provides us with noisy, the approximate examples of $f^*(\mathbf{x})$ evaluated at different training points. Each example $\mathbf{x}$ is accompanied by a label $y = f^*(\mathbf{x})$. The training examples specify directly what the output layer must do at each point x: it must produce a value that is close to y.

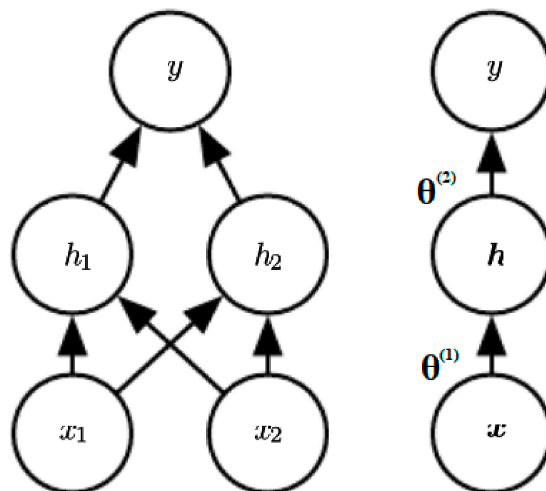
Considering the case of the daily stock price prediction at time t given its historical values with certain length $t - n$. During the training the output y is specified and known (daily price at time t) in order to find the optimal function f^* (illustrated in section 1.3 equation 1.1).

The behaviors of the other layers is not directly specified by the training data. The learning algorithm must decide how to use those layers to produce the desired output, indeed the training data does not say what each individual layer should do. The learning algorithm must decide how to use these layers to best implement an approximation of f^* because the training data does not show the desired output for each of these layers, they are called hidden layers (Goodfellow et al. 2016, p. 169).

The concept of hidden layer requires the choice of the activation functions that will be used to compute the hidden layer values. The activation function in a simple perceptron (Figure 2.1) represents the relation between input and output of each layer. This function helps the network to learn complex patterns in the data. In general they are non-linear functions as Sigmoid and Hyperbolic Tangent (section 2.3).

The Figure 2.2 provides an example of Deep neural network with one hidden layer composed of two units. Here, we have the a matrix of weights $\theta^{(1)}$ which describes the mapping from **x** to **h**, and the vector $\theta^{(2)}$ describes the mapping from **h** to **y**. In this case we have a model with function in form: $f^1(f^2(\mathbf{x}))$. When we have more layers, the different activation functions

Figure 2.2: Feedforward network example



Source: Goodfellow et al. (2016)

take the output value from neurons of the previous layer as an input with the relative weights θ, which mapping the influences. These weights must be learned from the algorithm to obtain the best approximation function: this operation is known as the learning process. In general each node of hidden layer can be explained with the following generic function:

$$h^{(m)} = f(x; \theta^{(m)}, b) \tag{2.3}$$

Where the variable m refers to the number of hidden layers. For example, in a deep learning model with two hidden layer, the $m = 1$ refers to the first hidden layer. Considering the Fig-

ure 2.1 and the Equation 2.3, the value of a layer (output or hidden), is given by an activation function which analyses the sum of inputs with their relative weights plus a specific term b.

The term b indicates the bias parameter of the affine transformation. It does not refer to the idea of statistical bias concept, which indicates that algorithm's estimations are not equal to the true values. The bias term (b) is a constant and represents an additional input into the next layer that will always have the value equal to 1. Bias units are not influenced by the previous layer (they do not have any incoming connections) but they do have outgoing connections with their own weights. The bias unit guarantees that even when all the inputs are zeros, there will still be activation in the neuron.

The activation function can also be linear, but in this case, the feedforward network as a whole would remain a linear function of its input. Hence, it is equivalent to a single layer network. We illustrated a general structure of deep neural network, but there are a lot of models with different architectures, types of neurons and links between them.

2.3 Activaction functions

The deep learning models try to find a non-linear relationship between input and output by identifying the function $f^*(x)$. There are various non-linear functions f which can be used for this purpose. In this section, we illustrate the functions that we chose for our algorithm.

2.3.1 Sigmoid function

The Sigmoid activation function, sometimes denominated logistic function, is used mostly in feedforward neural networks. It is a bounded differentiable real function, defined for real input values (Nwankpa et al. 2020). The Sigmoid function is:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.4}$$

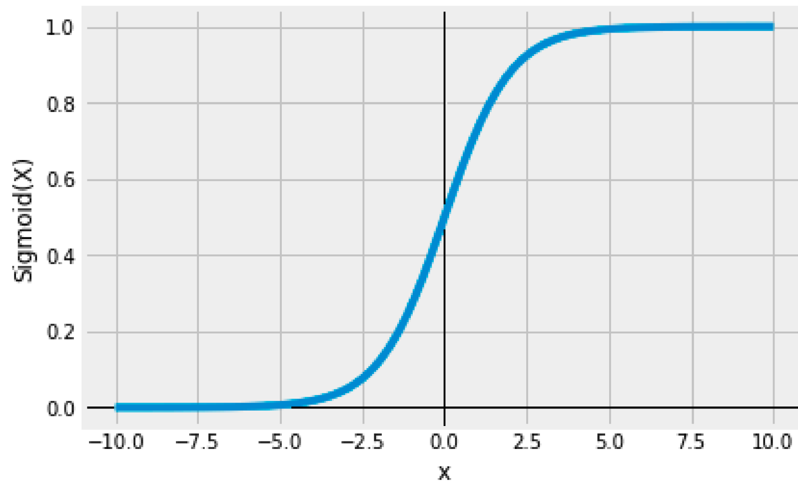
Figure 2.3 provides a graphical representation of this function. The range is between 0 and 1. The Sigmoid function appears in the output layers of the deep learning architectures, and is used to predict probability-based output. Sigmoid functions have been applied successfully in the binary classification problem. We note that the output values of this function are not zero centered.

2.3.2 Hyperbolic tangent function

The hyperbolic tangent function known as $Tanh$ function, is a smoother zero-centered function whose range lies between -1 to 1.

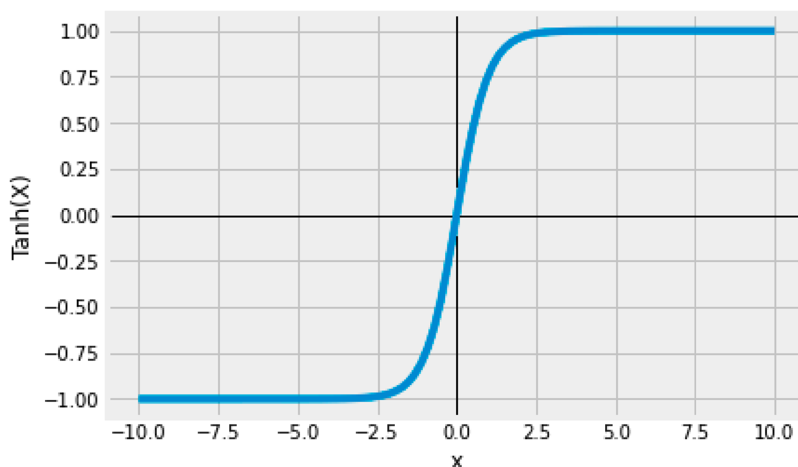
$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.5}$$

Figure 2.3: Sigmoid function



This function is used in hidden layer, the figure 2.4 provides a graphical representation.

Figure 2.4: Tanh function



These two kinds of functions are used for various deep learning models with different tasks. They represents the activation function that we will implement in our model presented in section 2.8. However, these nonlinear functions can lead to various problems during the optimization model, during the optimization to find the best parameters θ. These problems are discussed in the following section.

2.4 Model optimization

The deep learning model performance (P) represents an important element of analysis. If the performance is good, the model can identify the best approximation function f^*. A method for evaluating the model performance is represented by the loss functions, which calculate how the predictions deviated from the actual results. A common loss function is the Mean

Squared Error(MSE) and is given by the following equation:

$$MSE = \frac{\sum_{i=1}^{T}(y_i - y_i^{'})^2}{T} \tag{2.6}$$

where $Y = (y_1, y_2, \cdots, y_T)$ is the vector of actual observation, $Y^{'} = (y_1^{'}, y_2^{'}, \cdots, y_T^{'})$ is the vector of the model predictions, and T is the number of prediction generated. As shown in equation 2.3, the weights map the influence of the input on output: hence, we have to determine which are the best weights to obtain a good performance, which is associated with a lower loss function value.

The deep learning architecture involves many variables that make complicated the calculation of good weights for the model. Hence we introduce a procedure known as optimization: a searching algorithm that navigates the space of possible sets of weights that the model may use to make good or good enough predictions. We usually phrase most optimization problems to minimize the objective function, represented in our case by the loss function (also called error function or cost function).

For simplicity, considering the simple perceptron structure as defined in Figure 2.1, we can define the model as:

$$y_i^{\prime} = f(\mathbf{x}; \theta_i) \tag{2.7}$$

Where **x** is the input vector of the model, and θ represents the vector of weights for each input observation. Furthermore, let rewrite the loss function MSE as:

$$MSE = \frac{\sum_{i=1}^{T}(y_i - f(\mathbf{x}; \theta_i))^2}{T} \tag{2.8}$$

$$C = MSE = g(\theta_i) \tag{2.9}$$

The letter C indicates the loss (or cost) function. Our purpose is to minimize this loss function in relation to the parameters θ_i, the weight associated to each input x_i. We use the classical concept of derivative in order to identify the point of minimum. However, the machine learning model involves functions with multiple inputs (as Figure 2.2). In this case we must make use of the notion of partial derivative: $\frac{\partial}{\partial \theta_i} g(\theta)$ which measure how g changes as the variable θ_i changes while holding all the others constant. The vector that contains all the partial derivatives is called gradient ($\nabla_{\theta} g(\theta)$). To minimize our Loss function, we would like to find the direction in which g decreases fastest (Goodfellow et al. 2016, p. 85). To do so, we can use the directional derivative. The directional derivative in direction **u** (a unit vector) is the slope of the function f in direction u. The function minimization for the variables θ_i is:

$$\min_{\mathbf{u}, \mathbf{u}^{\top}\mathbf{u}=1} \mathbf{u}^{\top} \nabla_{\theta} g(\theta) \tag{2.10}$$

This is minimized when **u** points in the opposite direction as the gradient. The positive gradient points directly uphill and the negative gradient directly downhill. We can decrease

g by moving in the direction of the negative gradient, this is know with the name Gradient Descend. Hence, we move across different points of the loss function in order to reach the minimum. Let us introduce a general real-life example that can help us understand this concept: we can think of a blindfolded man at the top of a mountain and want to reach the mountain's lowest point. He will take a step and look for the valley's slope, whether it goes up or down. Once he is sure of the downward slope, he will follow that and repeat the step many times until he has descended completely.

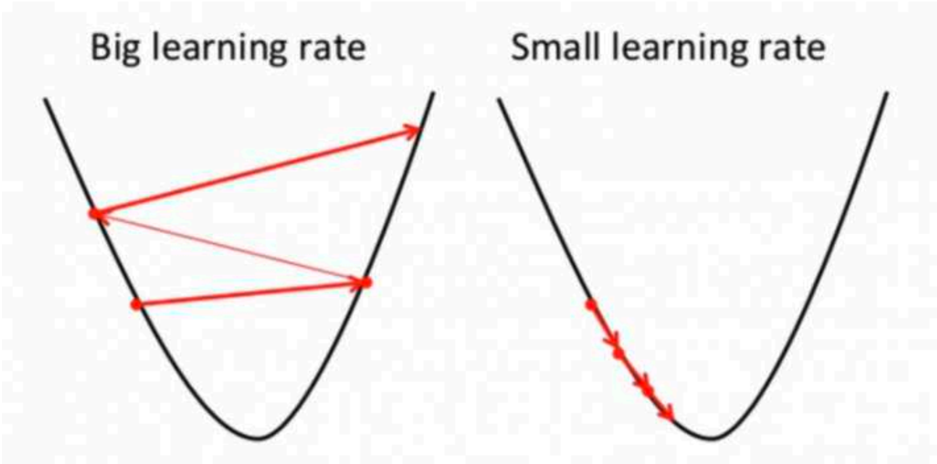
Hence, given the value of loss function with the initial weights, the gradient descent proposes the new point:

$$\theta' = \theta - \varepsilon \nabla_{\theta} g(\theta) \tag{2.11}$$

θ' indicates the new vector of weights and ε is the learning rate, a positive scalar that determines the size of the step. A popular approach is to set a small value of the learning rate. Indeed a high value of learning rates puts the model at risk of overshooting the minimum so it will not be able to converge; because with this learning rate we jump the minimum point (see figure 2.5). On the other hand, when the value is too small, the time required to learn the model increases.

Let see an example about gradient descent and the role of learning rate. Starting from the equation 2.6 which has the form of $Y = X^2$, an equation of parabola in Cartesian coordinates system. The objective of the gradient descend is to find the minimum point by moving in a small steps. Figure 2.5 illustrates the dynamic of this process; each arrow indicates the new point of the function reached with Equation 2.11 and the importance of setting a small learning rate. Considering that the Equation 3.3 can be function of more variables, for ex-

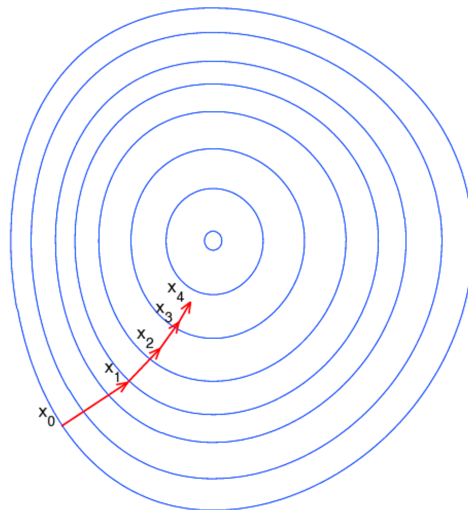
Figure 2.5: Gradient descent process



Source: Donges (2020)

ample the parameter θ and b (bias) as introduced in Equation 2.3 (section 2.2), we can also illustrate the gradient descent process though a contour plot of three dimensional function; as shown in Figure 2.6.

Figure 2.6: Gradient descent process: contour plot



Source: Donges (2020)

2.5 Back-propagation

When we use a feedforward neural network to process an input x and produce an output y', information flows forward through the network. The inputs x provide the initial information that then propagates up to the hidden units at each layer and finally produces the prediction y'. This is called forward propagation. Finally, we evaluate the model's performance with loss function (Equation 2.6). The back-propagation algorithm allows the information from the cost to then flow backwards through the network in order to compute the gradient (Goodfellow et al. 2016, p. 204).

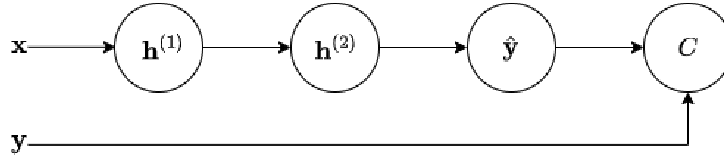
The gradient's analytical expression is straightforward, but the numerical evaluation of this expression can be computationally expensive. The back-propagation algorithm does so using a simple and inexpensive procedure.

The back-propagation purpose is to adjust each weight in the network in proportion to how much it contributes to the overall error. The "backward" part of the name stems from the fact that the gradient calculation proceeds backward through the network, with the gradient of the final layer of weights being calculated first and the gradient of the first layer of weights being calculated last. Partial computations of the gradient from one layer are reused in the gradient's computation for the previous layer.

For simplicity, we analyze how this method works in a simple neural network with two hidden layers. Figure 2.7 illustrates this simple network, in this case we set the loss function (Equation 2.6) as $C = g(\theta_i)$.

Where x represents the input dataset. As illustrated in section 2.2, each layer has a specific

Figure 2.7: Feedforward model with two hidden layers



activation function. In this case we have the following model's structure:

$$h^{(1)} = f^{(1)}(x; \theta^{(1)}) \tag{2.12}$$

$$h^{(2)} = f^{(2)}(h^{(1)}; \theta^{(2)},) \tag{2.13}$$

$$y' = \theta^{(3)} h^{(2)} \tag{2.14}$$

The term θ represents the input weights vector for each layer, the parameters that our model must learn. The objective of backpropagation is to find the value of θ, the various weights which minimize the loss function C (Equation 2.6).

Since the deep learning model has more than one layer, the loss function value depends on the weights value via a chain of many functions. Hence, we use the chain rule to compute the derivative of the loss function. The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known (Goodfellow et al. 2016, p. 205). For example, let x be a real number, and let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$, the chain rule states:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.15}$$

Each component of the derivative of C for each weight in the network can be calculated individually using the chain rule. However, it would be extremely inefficient to do this separately for each weight. The back-propagation algorithm involves first calculating the derivatives at the last layer (or output layer). These derivatives are an ingredient in the chain rule formula for layer $N - 1$, so they can be saved and re-used for the second-to-last layer. Thus, in back-propagation, we work our way backward through the network from the last layer to the first layer, each time using the last derivative calculations via the chain rule to obtain the derivatives for the current layer. In this way, the back-propagation algorithm allows us to efficiently calculate the gradient for each weight by avoiding duplicate calculations.

Given the deep learning model as illustrated in Figure 2.7, the back-propagation first calculates:

$$\frac{\partial C}{\partial \theta^{(3)}} \tag{2.16}$$

then:

$$\frac{\partial C}{\partial \theta^{(2)}} \tag{2.17}$$

and then:

$$\frac{\partial C}{\partial \theta^{(1)}} \tag{2.18}$$

We can express the loss function C explicitly as a function of all the weights in the network by substituting in the expression for each layer:

$$\begin{aligned} C(y, y') &= C(y; \theta^{(3)} h^{(2)}) \\ C(y, y') &= C(y; \theta^{(3)} f^{(2)}(\theta^{(2)} h^{(1)})) \\ C(y, y') &= C(y; \theta^{(3)} f^{(2)}(\theta^{(2)} f^{(1)}(\theta^{(1)} x))) \end{aligned} \tag{2.19}$$

Specifically, when we apply the chain rule for each layer of the model (Figure 2.7), we get the following equation for output layer:

$$\frac{\partial C}{\partial \theta^{(3)}} = \frac{\partial C}{\partial y'} \frac{\partial y'}{\partial \theta^{(3)}} \tag{2.20}$$

Next, we calculate the gradient of second hidden layer. Since C is now to steps away, we have to use the chain rule twice:

$$\frac{\partial C}{\partial \theta^{(2)}} = \frac{\partial C}{\partial y'} \frac{\partial y'}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial \theta^{(2)}} \tag{2.21}$$

Finally, we can calculate the gradient with respect to the weights of the first layer, this time using another step of the chain rule.

$$\frac{\partial C}{\partial \theta^{(1)}} = \frac{\partial C}{\partial y'} \frac{\partial y'}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\theta^{(1)}} \tag{2.22}$$

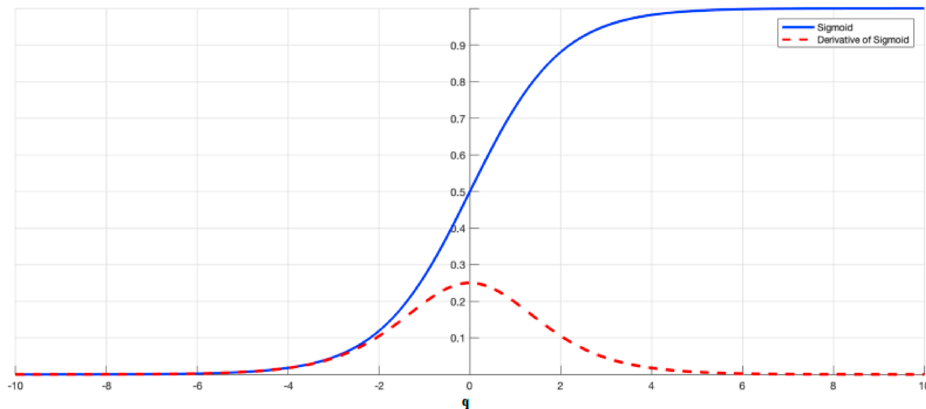
The first two terms in chain rule expression are shared with the second hidden layer's gradient calculation. Hence computationally, it is not necessary to calculate the entire expression.

The back-propagation refers to the method for computing the gradient, while other algorithms knowing as optimizer (see section 2.13), are used to perform learning using this gradient (Goodfellow et al. 2016, p. 204).

However, some drawbacks in gradient calculation must be considered. In section 2.3, we illustrate some examples of non-linear activation function. The back-propagation works with the derivative of the activation functions with respect to the weights θ, the variables which we want optimize. Figure 2.8 illustrates the shape of this derivative of the Sigmoid function (formula 2.4).

The activation function's value depends on the variable $q_i = \theta_i x_i$ where x_i represents the specific input observation with the associate weight. If the value of q is larger or smaller, the derivative has a value close to zero. This can cause problems during the application of the chain rule. The gradient decreases exponentially as we propagate down to the initial layers

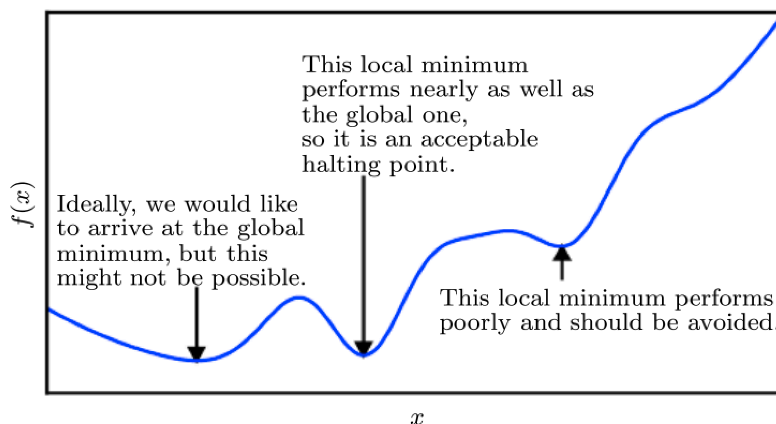
Figure 2.8: Derivative of Sigmoid function



and could vanish (become zero). This phenomenon is known as Vanish Gradient Problem.

Furthermore, in the context of deep learning, we optimize functions that may have many local minimum points that are not optimal and many saddle points surrounded by flat regions. When the input to the function is multidimensional, the optimization process is challenging. Therefore, we usually settle for finding a value of g (Equation 2.9) that is very low, but not necessarily minimal in any formal sense (Goodfellow et al. 2016, p. 84). An example of this problem is illustrated in Figure 2.9.

Figure 2.9: Global and local minimum of a function



Source: Goodfellow et al. (2016)

2.6 Overfitting

The main challenge of machine learning is that the model must perform well on new, previously unseen inputs—not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalization. During the training phase, we compute the loss function to measure the model's performance, which is also called the training error. However, we want to obtain a low generalization error (or test error) as well,

which is defined as the expected value of the error on new input (Goodfellow et al. 2016, p. 110).

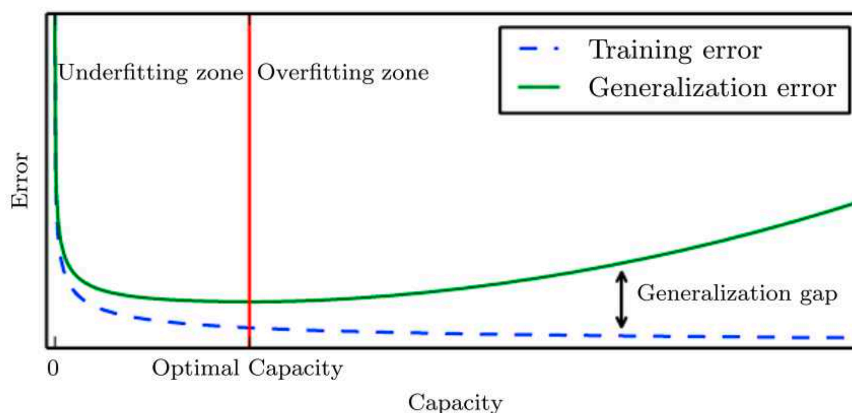
We typically estimate the test error by measuring the model performance on a test set of examples. The loss function can be the same as training error or different. Hence, the good performance of algorithm depends on its ability to make small the training error and the gap between test and training error. These two factors correspond to the two main challenges of machine learning. The first is called ”underfitting” and occurs when the model is not able to obtain a sufficiently low error value on the training set. On the other hand, in according to Salman & Liu (2019) and Goodfellow et al. (2016) we can define the second challenge as:

Definition 2.2. ***Overfitting:*** *The noise or random fluctuations in the training data is picked up and learned as concepts by the model. This results in a large gap between the training and test error.*

Deep Learning models are exposed to this problem because they have a large number of parameters that must be learned. However, the models are very flexible and we can adjust the various parameters to obtain a better performance. Parameters adjustment means that we can control the model by varying its capacity, defined as the ability to fit a wide variety of functions. When the capacity is appropriate, we can obtain a good model. A low capacity means that the model can struggle to fit the training set (underfitting). On the opposite, a high capacity model can overfit by memorizing properties of the training set that do not serve them well on the test set.

The Figure 2.10 provides a relation between error and capacity. We can observe that the

Figure 2.10: Underfitting vs Overfitting: Error analysis



Source: Goodfellow et al. (2016)

two errors follow different patterns. At the left of the graph, errors are very high and this is associated with an underfitting situation. When we increase the capacity of the model by varying some parameters, the training error decreases. However when we increase capacity, the gap between training and test error could increase. Furthermore, the size of the gap

can outweigh the training error reduction, resulting in the overfitting regime. For the deep learning model, applied to time series prediction in our case, we must pay attention to this problem: we can obtain a very good model in terms of training error, but this could be due to an overfitting regime. The researchers proposed various strategies to avoid this problem and try to stay in the optimal capacity zone. The chosen strategy for our model will be presented in section 2.14.

2.7 Recurrent neural network: architecture and optimization problem

Recurrent Neural Networks or RNNs are a family of neural network architecture for processing sequential data which measure a specific phenomenon over time or in a given order (of sequential events) without a concrete notion of time. These models represent an extension of the classical feedforward neural networks, presented in section 2.2, but with the implementation of feedback connections.

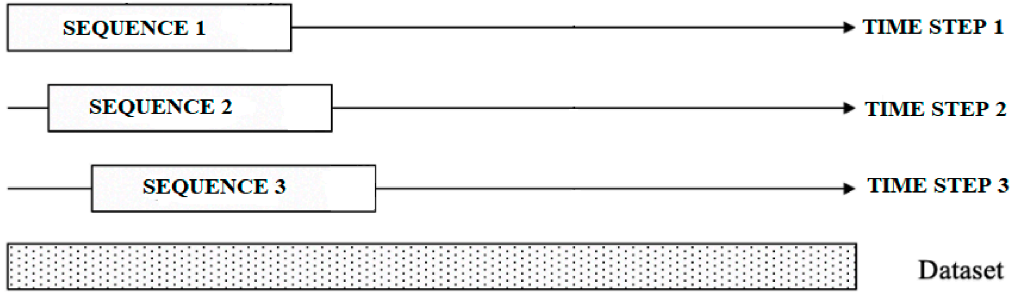
There are differences between the classical neural network and RNNs. The deep learning model, introduced in section 2.2 (Figure 2.2), processes the entire input dataset to obtain the output. In this case, the approach is different: we divide the input dataset into more sequences.

For example, consider a generic case where the input dataset x contains 200 sequential observations, e.g. the daily stock price for a specific time horizon. We divide this dataset into more subsequences with a specific length. For example, we can set a length of 50 to generate overlapping rolling window sequences. This overlapping process has the following structure: the first sequence represents the first 50 observations, the second contains the observations from 2 to 51. We continue to generate sequences with this length until the end of the dataset, where the last sequence contains the observations from 151 to 200. In this context, the input time step represents the portion of the dataset that is processed by the model. Figure 2.11 summarizes this process, the sliding window moves from the left to the right of dataset, and each increment represents the one sequence as described above.

The recurrent neural network processes the sequences of input data x_t, where t indicates the time steps, to obtain sequential output y' referring to time step t. The time step index refers to the sequence position: e.g., $t = 1$ represents the first sequence. The model time steps are represented by $t = 1, 2, \cdots, s$, where s is the total number of sequences in which we divided our model. The sequence of output, obtained from the various timestep, represents the vector $Y' = y'_1, y'_2, \cdots, y'_s$.

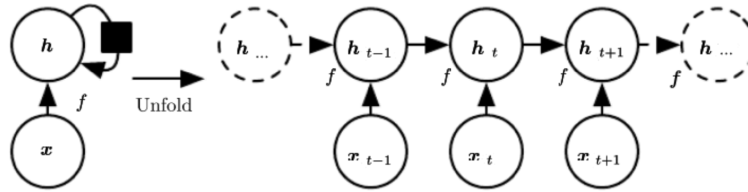
Given these aspects, we can adapt the hidden layer equation 2.3 in section 2.2 considering

Figure 2.11: Example of input sequences



Source: Ly et al. 2021

Figure 2.12: Recurrent neural network without output



Goodfellow et al. (2016)

the time step input. The value of the hidden layer neurons h_t of the RNN network is:

$$h_t = f(h_{t-1}, x_t; \theta) \tag{2.23}$$

The function f is non-linear as in equation 2.3. However, equation 2.23 contains the time step term t. In latter case, the value of the hidden layer does not depend only on the input data and the relative weights as in the feedforward model but also on its previous value h_{t-1}: the value of the previously hidden layer obtained when the model processed the previous sequence. When the model processes the first sequence of input $t = 1$, $h_{t-1} = h_0$ does not exist because there are not previous sequences processed by the model ($h_0 = 0$). In this case, the equation 2.23 becomes:

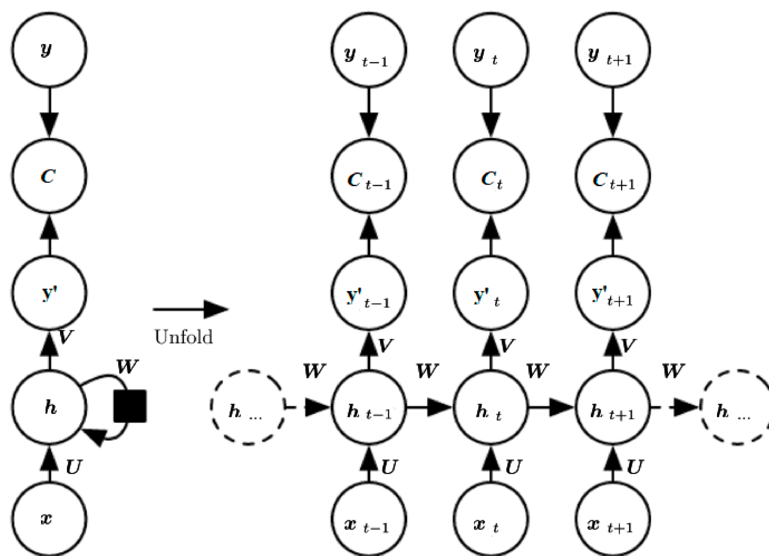
$$h_1 = f(x_1; \theta) \tag{2.24}$$

Figure 2.12 illustrates an example how the hidden layer of recurrent network works to process the various input sequences. The recurrent network processes information from the input time step x_t by incorporating it into the state h_t that is passed forward to other hidden layers through the various sequences. During the training, if the recurrent network performs a task that requires predicting the future from the past, the network typically processes the input sequences but considering the information stored in the previously hidden layer. This information represents a summary of the relevant information contained in the last input sequence for output prediction. Depending on the training criterion, this summary might selectively keep some aspects of the past sequence with more precision than other aspects. Hence, recurrent neural network performance implies the use of memory to store information and learn

how to select the information that has to be maintained in memory.

The purpose of this model is to unveil and learn the dependencies among various time steps. Thus, this algorithm can be used for predictions and is capable to analyze financial time series data (Fischer & Krauss 2017, p. 1). A complete structure of RNNs is summarized in Figure 2.13. y' is the output, and C represents the loss function (equation 2.9) which measures how far each y' is from the corresponding training target y. The network has input to hidden connections parametrized by a weight matrix **U**, hidden-to-hidden recurrent connections parametrized by a weight matrix **W**, and hidden-to-output connections parametrized by a weight matrix **V**. There are many ways to build an RNN with different functions and links between the cells. Figure 2.13 represents only one kind of RNN's architecture; we can alternatively use the information contained in the output (y'_t or y_t) and incorporates them in h_{t+1} instead of using the value of the previously hidden layer.

Figure 2.13: Recurrent neural network



Goodfellow et al. (2016)

However, the gradient computation, introduced in section 2.4, for the RNN networks uses a specific type of backpropagation algorithm: the backpropagation through time (BPTT). This is because the network contains various inputs sequences which must be processed, as shown in Figure 2.13. We calculate the gradient with the chain rule for each time step. The procedure is the same as the feedforward model presented in section 2.4, but with the difference that in feedforward case there is no share of information across the layers, while RNN works with more sequences that share information h_t. For simplicity, considering a RNN network with the input timestep $t = 3$. When we apply the process explained in section 2.5, we have to consider that h_3 is influenced by h_2, which is influenced by h_1 (see Figure 2.13). Hence, we back-propagate gradients through layers and also through time.

This difference is crucial because it makes complicated the backpropagation algorithm application. By the chain rule, each layer's derivatives are multiplied down the network (from

the final layer to the initial) to compute the derivatives of the initial layers. Applying the backpropagation to RNN cause a gradient propagation over many stages (due to the time step sequences), so it tends to either vanish or rarely explode with adverse consequences for optimization (Goodfellow et al. 2016, p. 401). The vanish problem depends on the fact that the hidden layers use a non-linear activation function and their derivatives calculated in each layer have low values. These derivatives are multiplied together, thus, the gradient value decreases exponentially as we propagate down to the initial layers. Considering that we apply this procedure to various time steps, we can face a situation where the gradient will go to zero exponentially fast. Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function (Goodfellow et al. 2016, p. 290). These problems can result in bad weights, which make challenging the training of the model and lead to a bad performance.

Another problem is that the RNN's model stores only the previous hidden layer information, which results in a short memory. Hence the model cannot be able to learn long-term relations from the different sequences. These problems have represented a challenge for various researchers who have proposed different solutions. A famous architectural approach that can solve this problem is called Long short-term memory. It is one of the most advanced deep learning architectures for sequence learning tasks, such as handwriting recognition, speech recognition, machine translation, and time series prediction (ibid., p. 410). In the recent years various LSTM model for stock price prediction were proposed: Fischer & Krauss (2017), Ta et al. (2020) and Hao & Gao (2020). The next sections explain LSTM algorithm structure and its application for stock price predictions.

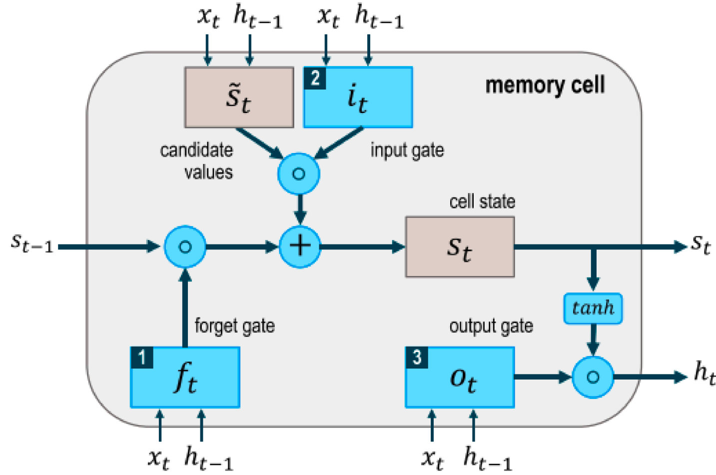
2.8 LSTM algorithm

Long short-term memory networks are specifically designed to learn long-term dependencies and can overcome the previously inherent problems of RNNs. This model is introduced by Hochreiter & Schmidhuber (1997) and refined in the following years by various studies as in Gers et al. (2000). Instead of a unit of the hidden layer that applies a non-linear function when the information is stored and passed to the next sequences (or time steps) hidden layer unit, LSTM recurrent networks have LSTM cells that have an internal recurrence (a self-loop), which are able to store information and analyzes long-term dependencies across sequences.

LSTM networks are composed of an input layer, hidden layers, and an output layer as the classic feedforward model. However, in this architecture, the cell unit (or neurons in feedforward model) does not contain a simple activation function but presents a very complex structure as shown in Figure 2.14 (Fischer & Krauss 2017).

Each cell has the same inputs and outputs as an ordinary recurrent network but has more parameters and a system of gating units that control the flow of information (Goodfellow et al. 2016, p. 410). These memory cells or cells state are controlled and adjusted with three gates:

Figure 2.14: Structure of LSTM memory cell



Fischer & Krauss (2017)

a forget gate (f_t), an input gate (i_t) and an output gate (o_t). At each time step, t, these three gates act as filters with different purposes on information incorporated in the input analyzed by the memory cell. In general:

- The forget gate specifies which information is removed from cell state.
- The input gate describes which information is added to the cell state.
- The output gate defines which information from the cell state is used as output.

These gates are similar to a layer or a series of matrix operations, which contain different individual weights. In according to Hochreiter & Schmidhuber (1997), Gers et al. (2000), Graves (2013), Le et al. (2019) and Fischer & Krauss (2017) we can describe this model and its gates with the following notation:

- x_t is the input at timestep t.
- s_t and $\bar{s}_t$ are vectors for the cell states and candidate values.
- f_t, i_t and o_t are vectors for the activation values of the respective gates.
- h_t is the vector of the output of LSTM layer.
- $W_{f,x}$, $W_{f,h}$, $W_{\bar{s},x}$, $W_{i,x}$, $W_{i,h}$, $W_{o,x}$ and $W_{o,h}$ are weight metrices.
- b_f, $b_{\bar{s}}$, b_i, and b_o are bias vectors.

The first step determines which information should be removed from the previous cell state. Therefore, the activation values f_t takes the output of the last LSTM unit (h_{t-1}) and the current input x_t. The $sigmoid$ function scales all activation values into the range between 0 (completely forget) and 1 (completely remember).

$$f_t = sigmoid(W_{f,x}x_t + W_{f,h}h_{t-1} + b_f) \tag{2.25}$$

In the second step, the LSTM decides which new input information should be added to the network's cell states (s_t). The procedure comprises two operations. First, the candidate value must be computed through the $tanh$ function, which gives weight to the values passed by, deciding their level of importance (-1 to 1). Second, a $sigmoid$ layer decides whether the new information (or candidate value $\bar{s}_t$) should be updated or ignored (0 or 1). The equations of this step are:

$$\bar{s}_t = tanh(W_{\bar{s},x}x_t + W_{\bar{s},h}h_{t-1} + b_s) \tag{2.26}$$

$$i_t = sigmoid(W_{i,x}x_t + W_{i,h}h_{t-1} + b_i) \tag{2.27}$$

The new cell states s_t are calculated based on the results of the previous two steps with the symbol $\circ$ denoting the Hadamard product:

$$s_t = f_t \circ s_{t-1} + i_t \circ \bar{s}_t \tag{2.28}$$

In the last step, the h_t memory cell output is based on the output cell state but is a filtered version. First, we pass the previous hidden state and the current input (h_{t-1} and x_t) into a sigmoid function. Then the output of the sigmoid gate (o_t) is multiplied by the new values created by the $tanh$ layer from the cell state s_t (Le et al. 2019, p. 9).

$$o_t = sigmoid(W_{o,x}x_t + W_{o,h}h_{t-1} + b_o) \tag{2.29}$$

$$h_t = o_t \circ tanh(s_t) \tag{2.30}$$

The LSTM architecture allows to accumulate information over a long duration. Thanks to the forget gate (f_t), the model is able to control the flow of information of the previous input sequences which are processed. The algorithm manages an huge amount of information through various gates and relations derived from the input time steps in order to obtain the predictions. The information control allows to mitigate the vanishing gradient problem (Goodfellow et al. 2016, pp. 413–420).

2.9 Stock return prediction model

We built a deep learning model with an LSTM network for the daily adjusted-close stock price prediction. In this way, we can understand the daily stock movements to construct a specific investment strategy (see Chapter 3 section 3.4). The following sections presents the architecture and all the different strategies for our algorithm optimization and performance improvement.

2.10 Hardware

Neural networks usually involve large and numerous buffers of parameters, activation values, the calculation of gradient descend and backpropagation. All these computations must be completely updated during every step of training. These buffers are large enough to fall outside the traditional desktop computer's cache, so the system's memory bandwidth often becomes the rate-limiting factor. However, the introduction of Graphics processing units (GPUs) helps to solve these problems.

The GPUs are specialized hardware components that were initially developed for graphics applications and offer a compelling advantage over CPUs due to their high memory bandwidth. It breaks complex problems into thousands or millions of separate tasks and works them out at once. Considering that the Neural network training algorithms typically do not involve much branching or sophisticated control, they are appropriate for GPU hardware. Since neural networks can be divided into multiple individual ”neurons” that can be processed independently from the other neurons in the same layer, neural networks easily benefit from the parallelism of GPU computing (Goodfellow et al. 2016, p. 445).

For these reasons, we decided to use the hardware GPU provided by Colab, which offers a different models as NVIDIA K80 with 12 GB RAM. Running and optimizing a deep learning model with a dataset of 15-year stock requires much memory, and we can perform our algorithm faster with the GPU. Instead, CPU takes up a lot of memory and time to train this type of model.

2.11 Hyperparameters selection

The deep learning algorithms have several settings that we can use to control their performance. These settings are called hyperparameters. Bengio (2012) defines a hyperparameter for a learning algorithm A as a variable to be set prior to the actual application of A to the data, one that is not directly selected by the learning algorithm itself. Setting the correct value of hyperparameters is crucial to improve the capacity of the algorithm. The main hyperparameters which we must set in a deep learning algorithm are:

- **Minibatch (B):** Processing the entire dataset can require a considerable amount of time. Hence, we can divide the training dataset into more samples (also called minibatches).
- **Minibatch size:** Indicates the size of samples in which we have divided the entire dataset. For example, a minibatch size equal to 2 indicates that we divide the dataset into two samples, processed in parallel by the algorithm. This hyperparameter controls the number of training samples to work through before the model's internal parameters are updated. The minibatch size is typically chosen to be a relatively small number

of examples, ranging from 1 to a few hundred (Goodfellow et al. 2016, p. 152). A larger value yields faster computation (with appropriate implementations) but requires visiting more examples to reach a fair value of error. In theory, this hyperparameter should impact training time and not so much test performance. A value of minibatch size equal to 32 is considered the default value for the basics computations (Bengio 2012).

- **Epoch:** Indicates that the entire dataset is processed by the algorithm once, considering the output calculation and the optimization phase (section 2.4). Usually, the number of epochs is set greater than one: the algorithm processes the dataset several times to decrease the loss error function and to find the best weights. A correct epochs number is crucial to avoid overfitting.

When the training dataset is divided in more samples (B) with a defined size, the number of the minibatch needed to complete one epoch is called iteration. We can obtain this number by dividing the size of dataset for the size of minibatch.

Setting the appropriate hyperparameters is crucial to obtain a good model in terms of capacity. In particular, we have to pay attention to the number of epochs. A huge number of epochs means that we train our model many times. We can obtain a very low value of training error, but the algorithm could present an overfitting problem (Section 2.10).

2.12 Feature scaling

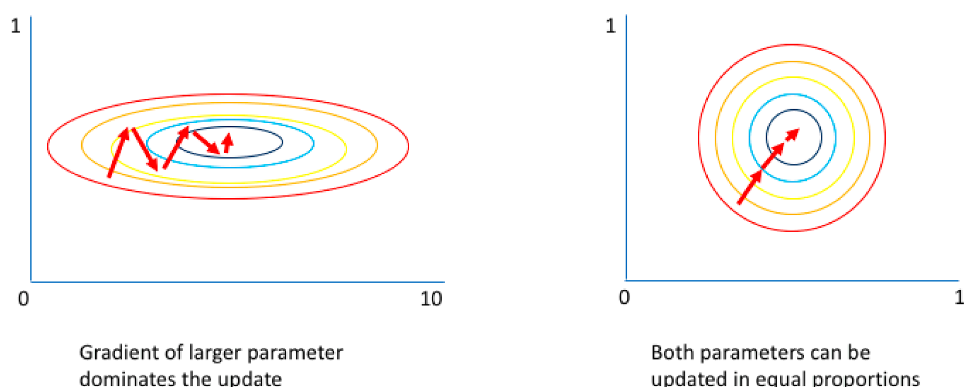
One preliminary process of managing the dataset to improve the model performance is known as feature scaling: the normalization of input dataset range. This process improves model performance, given its effect on gradient descent. Recall Figure 2.6 (section 2.4) which explains the gradient descent process in a simple case with a contour plot. Considering the loss function Equation 2.6 introduced in section 2.4, the output of deep learning model $Y' = (y_1', y_2', \cdots, y_T')$ can have different scale. The contour plot's topology of the loss function is different in relation to the scale. For example, Figure 2.15 shows the difference between a loss function contour plot when data are normalized with respect to unnormalized data. We can see that the gradient descent reaches the point of minimum in a more clear way with normalized data.
Considering our task: the daily stock price predictions. The stock price in absolute term could have big variations across the time. For this reason is better to consider the simple return as input dataset, which can be defined as:

Definition 2.3. ***Simple Return***

$$R_{t,m}^s = \frac{P_t^s}{P_{t-m}^s} - 1 \tag{2.31}$$

Where P_t is the price of the stock at time t. Figure 2.16 shows the difference in comparison between the adjusted-close stock price and simple return of four stock of the S&P500

Figure 2.15: Effect of data normalization on gradient descent

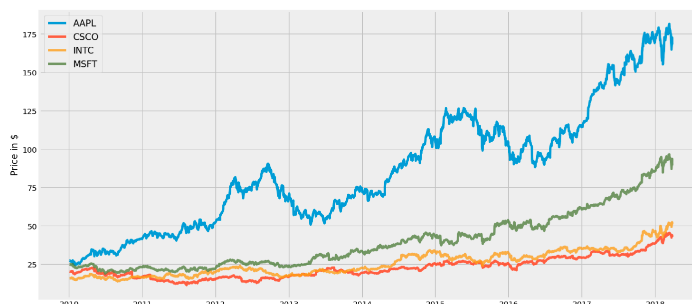


Source:Jordan 2018

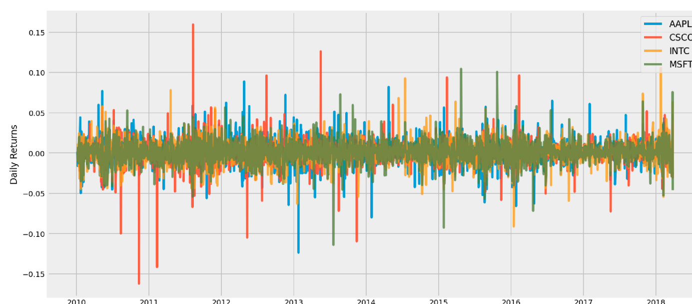
information technology sector: Apple (AAPL), Cisco (CSCO), Microsoft (MSFT) and Intel (INTC). The prices are collected from Yahoo finance website, from January 2010 to December 2019. As we can see, the stock prices of Apple (the blue line) are very high with

Figure 2.16: Stock price and simple return comparison

(a) Stock prices



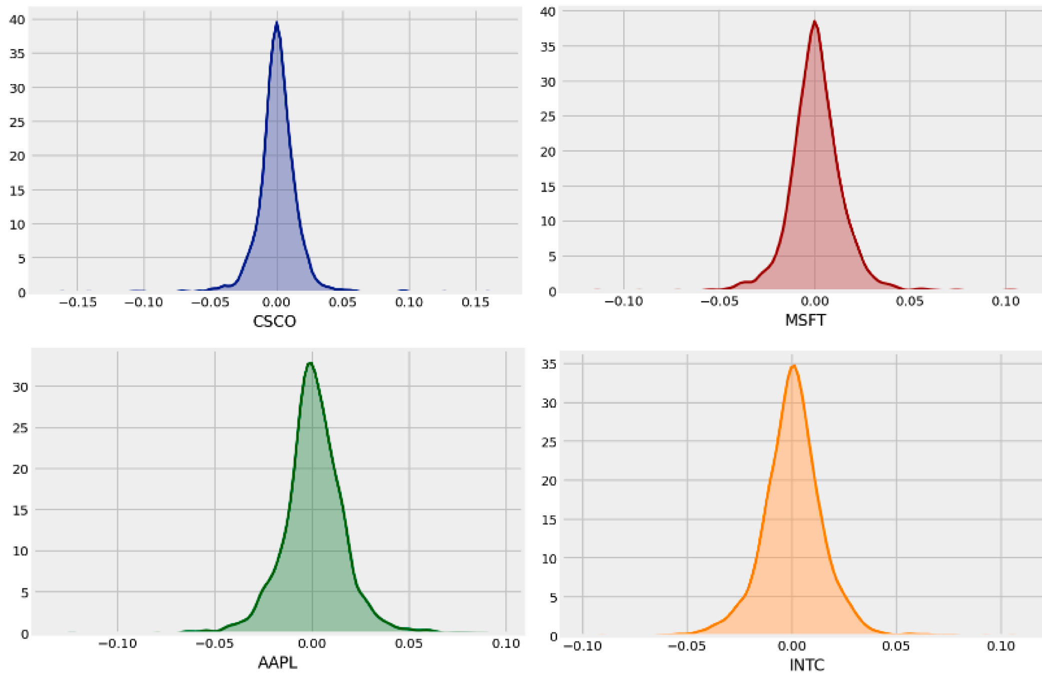
(b) Simple stock return



Source: Yahoo finance website

respect to Cisco (orange line), making the comparison of stock performance difficult. On the other hand, Figure 2.16 shows that the simple return make this comparison easier. Figure 2.17 illustrates a comparison of the simple return distribution between Apple, Cisco, Intel and Microsoft. From algorithmic point of view, the absolute price cannot be used as input because the range variations make difficult the gradient descent optimization. However, the simple return does not overcome this problem because the range is different across time (see

Figure 2.17: Simple return distributions



Source: Historical data obtained from Quandl dataset

figure 2.16).

The solution for this problems it is data normalization, which provides a dataset in a specific range and improves the algorithm performance. There are two main types: the Min-Max scaling and Z normalization (standardization).

Definition 2.4. *The Min-Max scaling is a scaling technique in which values are shifted and rescaled to end up ranging between 0 and 1. The formula of this technique is:*

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \tag{2.32}$$

Where X_{max} and X_{min} are respectively the maximum and the minimum values of the feature (in our case, the stock price or the simple return). The X' indicates the rescaled value.

Definition 2.5. *The Z normalization or standardization where the values are centered around the mean with a unit standard deviation in accordance with the following formula:*

$$X' = \frac{X - \mu}{\sigma} \tag{2.33}$$

Where μ is the mean of feature value and σ is the standard deviation of the feature value. This means that the attribute's mean becomes zero and the resultant distribution has a unit standard deviation.

There is no general rule for the best feature scaling technique. In according with Fischer & Krauss (2017) when we consider the stock return it is better to apply the Z normalization. However, we can also consider the stock price as in Ta et al. (2020) or Hao & Gao (2020)

and apply the Min-Max scaling to scale the historical data in a specific range to compare different asset. The learning rate of the gradient descend benefits from these scaled ranges of data and this improves the speed and accuracy of the model.

2.13 The optimizer: ADAM

In section 2.4, we have introduced the gradient descent algorithm, which tries to minimize the model's loss function and find the best weights. One common problem of algorithms is that they work with large and computationally expensive datasets. Furthermore, the deep learning algorithms have a very complex structure (e.g., the LSTM memory cell in our model). For this reason, nearly all these algorithms are powered by one very important extension of gradient descent and back-propagation (section 2.5): the Stochastic Gradient Descent or SGD. This algorithm attempts to find the global minimum by adjusting the network configuration after each training point. Instead of finding the gradient for the entire data set, this method merely decreases the loss function by approximating the gradient for a randomly selected batch. Specifically, on each step of the algorithm, we can sample a minibatch of examples $B = x(1), ..., x(m')$ drawn uniformly from the training set. The minibatch size m' is typically chosen to be a relatively small number of examples, ranging from 1 to a few hundred (Goodfellow et al. 2016, p. 152). The SGD gradient is given by:

$$g = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} L(x^i, y^i, \theta) \tag{2.34}$$

Where θ represents the vector of the weights, C the loss function (equation 2.6), x the input and y the output. The gradient downhill with learning rate e in this case is:

$$\theta \leftarrow \theta - eg \tag{2.35}$$

However, in practice, the learning rate gradually decreases over time: it is denoted by e_k. We impose this condition because the SGD algorithm introduces a noise source with the random sampling of m' training examples. Figure 2.18 illustrates how the SGD works.

Figure 2.18: SGD optimizer

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i C(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

Goodfellow et al. (2016)

The learning rate determines the step size at each iteration while moving toward a minimum of a loss function. A benefit of stochastic gradient descent is that it requires much less computational time than true gradient descent while still generally converging to a minimum (although not necessarily a global one). Hence, this method is handy for complex deep learning models that process a huge amount of data. Furthermore, there are various optimizer algorithms based on SGD, which are well suited for LSTM architecture. In this work, we decided to use the ADAM optimizer, which combines the best properties of various approaches (like RSMprop that works well in on-line and non-stationary settings). This model is proposed by Kingma & Ba (2014) and the name derives from the phrase "adaptive moments" (Goodfellow et al. 2016, p. 308). Let $C = g(\theta)$ our loss function, as introduced in Equation 2.9 in section 2.4, which we want minimize. However, $g(\theta)$ is a stochastic scalar function in this case, that is differentiable with respect to the parameters θ (the input weights as explained in section 2.2). The stochasticity might come from the evaluation at random sub-samples, as the minibatch or subset of the training dataset (as for SGD optimizer). The equation $j_t = \nabla_\theta g_t(\theta)$ denotes the gradient with respect to θ evaluated at timestep t. According to Kingma & Ba (2014), the algorithm updates exponential moving average m_t of the gradient (or first moment) and the squared gradient (or second moment), which are defined by:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) j_t \tag{2.36}$$

$$\nu_t = \beta_2 \nu_{t-1} + (1 - \beta_2) j_t^2 \tag{2.37}$$

Where the hyperparameters $\beta_1, \beta_2 \in [0, 1)$ control the exponential decay rates of these mov-

Figure 2.19: ADAM optimizer

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $g(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$j_t \leftarrow \nabla_\theta g_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot j_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot j_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Goodfellow et al. (2016) and Kingma & Ba (2014)

ing average. The moving averages themselves are estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradient. However, these moving averages are initialized as vectors of zero, leading to moment estimates that are biased towards zero, especially during the initial timesteps, and especially when the decay rates are

low: for example, when the β_1 and β_2 are close to 1 (Kingma & Ba 2014, p. 2). To solve this problem, the authors decide to counteract these biases by computing bias-corrected first and second-moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.38}$$

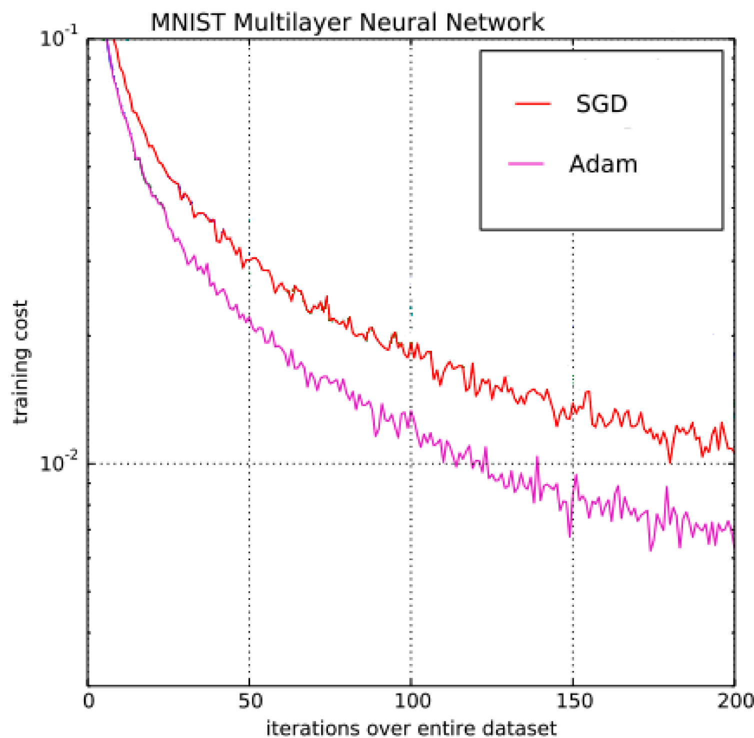
$$\hat{\nu}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.39}$$

These estimators are used to update the parameters which yield the ADAM update rule (equation 2.40). α indicates the learning rate as introduced in section 2.4, while ε represents a small constant for numerical stabilization.

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{\hat{\nu}_t} + \epsilon} \hat{m}_t \tag{2.40}$$

Figure 2.19 summarizes the ADAM optimization algorithm. The good default value for the principal hyperparameters, according to Kingma & Ba (ibid.) are: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$ and $\alpha = 0.001$ The ADAM optimizer is implemented through the Keras API,

Figure 2.20: Adam vs SGD: cost function



Kingma & Ba (2014)

which provides the following default parameters: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1e - 08$. We can observe that they are the same values as those proposed by Kingma & Ba (2014). According to the authors, ADAM is characterized by various benefits as computational efficiency and little memory requirement. It is well suited for large problems in

terms of data/parameters.

Figure 2.20 shows a comparison between the two optimizer algorithms, provided by Kingma & Ba (2014) illustrated in this section. These comparison is based on a multi-layer neural network model with two fully connected hidden layers with 1000 hidden units each, the task is to classify the handwritten digits of the MNIST dataset, a database of handwritten digits, provided by LeCun & Cortes (2010), which has a training set of 60,000 examples, and a test set of 10,000 examples We can see that ADAM makes faster progress for loss function minimization in terms of the number of iterations (ibid.). Given all these benefits for efficiency and cost function minimization, we decide to use ADAM for optimizing the deep learning algorithm.

2.14 Regularization strategies: dropout and early-stopping

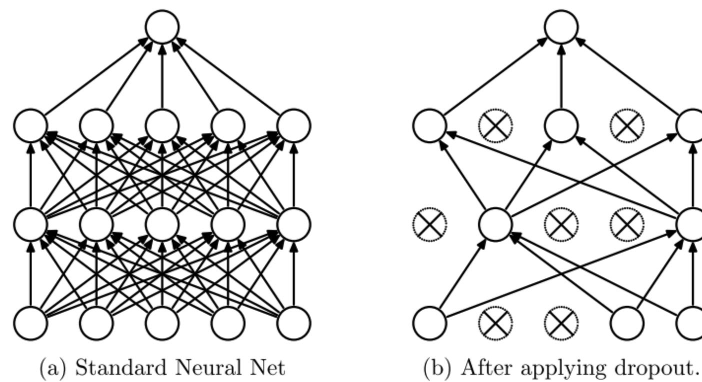
In section 2.6 we have introduced the concept of underfitting and overfitting (definition 2.2), based on the gap between training and test error. A good algorithm must avoid these problems and should perform well with the new inputs. To pursue this goal, we have to pay attention to the model's architecture, particularly to the number of hidden layers and neurons. The LSTM cell has a complex architecture; hence a huge number of cell (also called neurons) can create a sophisticated model which can lead the algorithm to overfitting zone. There is no general rule about the optimal number of neurons for a specific dataset. Our model's input dataset contains 15 years daily adjusted-close stock prices, which provides a significant number of observations to train the algorithm. We decided to consider a structure with a relatively low number of neurons and layers. This choice is based on various model architectures illustrated in different studies about LSTM for stock price prediction. We try to set different LSTM architecture to understand how to lose function of test and training set varies (see section 3.1 in chapter 3).

Our purpose is to create a model which can obtain a good prediction with a relatively low number of epochs to avoid the overfitting zone as shown in Figure 2.10 in section 2.6. However, a simple structure is not the only way to avoid overfitting. Various strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These different types of strategies are known as Regularization: some of them put extra constraints on a machine learning model, such as adding restrictions on the parameter values (Goodfellow et al. 2016, p. 228). There is no best form of Regularization, but we must choose a well-suited form for the particular task we want to solve.

The first regularization strategy implemented is called dropout introduced by Srivastava et al. (2014). The term dropout refers to dropping out units (hidden and visible) in a neural network. By dropping a unit out, we mean temporarily removing it from the network, along with

all its incoming and outgoing connections. For simplicity, Figure 2.21 shows an example of dropout regularization in classical feedforward networks. In the neural networks, which are

Figure 2.21: Dropout model



Dropout Neural Net Model. **Left**: A standard neural net with 2 hidden layers. **Right**: An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.

Srivastava et al. (2014)

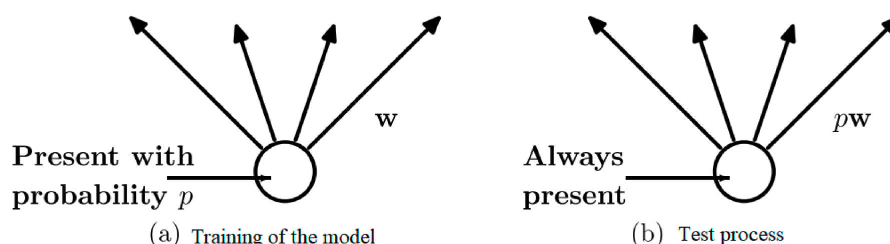
based on non-linear relationship, we can effectively remove a unit from a network by multiplying its output value by zero (Goodfellow et al. 2016, p. 258). The choice of which units to drop is random.

The Keras API easily implements the dropout by randomly selecting nodes to be dropped-out with a given probability each weight update cycle. We can set the rate, which represents the fraction of the input unit to drop. According to the other LSTM model proposed in Fischer & Krauss (2017), Ta et al. (2020) and Hao & Gao (2020) the dropout will be applied to each hidden layer with varying rate: e.g. 0.1, 0.05, 0.2.

Hence, when the dropout is applied, the model will result in a thinned network with the units survived. The model obtained is a subset of different networks, and for these reasons, it becomes less sensitive to a specific weight of neurons. In our case, this means that we take off some LSTM memory cells from the hidden layer randomly. So training a neural network with dropout can be seen as training a collection of different thinned networks, this process is applied during the training phase. Regarding the test phase, we can use a simple approximate average method, as shown in Figure 2.22. The idea of dropout is the following: during the training, a specific unit is present with probability p, and it is connected with the units of the next layer with a vector of weights ($\mathbf{w}$). At test phase, the units (neurons) are always present, but their weights are multiplied by p.

The dropout has two principal advantages: the first is that this technique is computationally cheaper and it does not significantly limit the type of model or training procedure used. However, we have to pay attention: the dropout is a regularization technique and reduces the effective capacity of a model. To offset this effect, we must increase the size of the model. Typically the optimal test set error is much lower when using dropout, but this comes at the

Figure 2.22: Dropout during the training and test phases



Srivastava et al. (2014)

cost of a much larger model and many more iterations of the training algorithm (Goodfellow et al. 2016, p. 265).

The second regularization strategy that we applied to our model is called early-stopping. Considering Figure 2.10 which describes the relation between model capacity and the two kinds of errors, we often observe that training error decreases steadily over time, but test error begins to rise again. The deep learning model processes the entire dataset and optimizes its weights through the backpropagation (see section 2.4) during each epoch. The backpropagation algorithm searches the new weights which minimize the loss function and this increases the capacity of the model (Figure 2.10 section 2.6). Hence, it is necessary to understand the evolution of the training error loss function (Equation 2.6 section 2.6) across the epochs to obtain information about the model capacity. If the loss function value does not decrease, the model capacity does not increase.

The early-stopping tries to stop the training when there is no improvement of model capacity across the various epochs. Every time that the error on the training set improves, we store a copy of the model wights finding during the backpropagation phase. The algorithm terminates when there is no improvement of the training error for some pre-specified number of epochs, which we can set arbitrarily. The goal is the identification of the number of epochs associated with low value of test error.

The early-stopping is a very unobtrusive form of regularization, in that it requires almost no change in the underlying training procedure, the objective function, or the set of allowable parameter values (ibid., pp. 246–248). Following the API Keras, this strategy in practice stops the model's training when a monitored loss function (e.g. Mean Squared Error Equation 2.6) has stopped improving. We can decide the number of epochs with no improvement after which training will be stopped. This last parameter is called patience. For example, if we set patience equal to 10 ($\psi = 10$) the training will stop after ten times with no improvement in the loss function. When the training algorithm terminates, we return the weights at epoch $\psi - 10$ rather than the latest, because this is the last epoch that provided a model capacity improvement.

These strategies can help to avoid overfitting and good model performance.

Chapter 3

Model results and investment strategy

3.1 Model set up

This section presents our deep learning model with LSTM architecture for daily adjusted-close price prediction.

We collected the historical daily adjusted-close stock prices, from January 2005 to December 2019, of 500 large-capitalized companies (Standard & Poor's 500) listed on American Stock Exchange for the model set-up. These data are collected from Yahoo finance website.
For the model configuration, we used Google Colab: a Python 3.9 cloud facility, which is especially well suited to machine learning and data analysis. We built the algorithm configuration with Keras, an API designed for deep learning model. Each stock contained 3775 observations.

The Training-Test split ratio has been set at $80 : 20$, which represents the classical proportions (Goodfellow et al. 2016, p. 121). Hence, we obtained a training dataset of 3020 observations and a test dataset of 755 observations (three trading year) for each stock.

The first step was to normalize these data for a better model optimization as introduced in section 2.12: we decided to use the Min-Max scaled (see definition 2.4). Given this normalized dataset, we defined the input time step (introduced in section 2.7). Recall that the LSTM model processes the input dataset in various sequences (time steps). We decided to set two kinds of sequences:

- Length of 90 days: the price of a stock in a t day is based on past prices sequences until $t - 90$ (three previous months).
- Length of 240 days: comprising the information of approximately one trading year.

We compared the algorithm performance in relation of these two types of sequences. As in the example illustrated in Figure 2.11 (section 2.7), we generated overlapping rolling window sequences. We obtained the daily prices forecasted on the basis of with the previous 90 days (first case) or the previous 240 days. The LSTM model was applied to each constituent of S&P500, taking into account that they changed across time. Hence we could have a very different amount of data for each constituent when we train the model. Given our model's complexity to learn dependencies between sequential data, this could lead to a different performance concerning companies' presence in S&P500. For this reason, we decided to select

Figure 3.1: Time steps (or sequences) of the input dataset

(a) Length of 90 days

Input vector for a given Stock S

Date	1	2	3	4	...	87	88	89	3020
S	Normalized adjusted-close price								

First Sequence

Date	1	2	3	4	...	87	88	89	90
S	Normalized adjusted-close price								

Second Sequence

Date	2	3	4	5	...	88	89	90	91
S	Normalized adjusted-close price								

(b) Length of 240 days

Input vector for a given Stock S

Date	1	2	3	4	...	87	88	89	3020
S	Normalized adjusted-close price								

First Sequence

Date	1	2	3	4	...	87	88	89	240
S	Normalized adjusted-close price								

Second Sequence

Date	2	3	4	5	...	88	89	90	241
S	Normalized adjusted-close price								

from the constituents listed in December 2019 only those with historical data from January 2005 and excluded the companies that were listed after this period. This is a relevant disadvantage (we cannot include Facebook, for example) but could help to build a model where each stock has the same dataset length.
The hyperparameters selection is a fundamental operation to obtain a good model performance. Various researchers proposed an LSTM approach for daily stock price prediction with specific hyperparameters, for example:

- Fischer & Krauss (2017) presents a model with one LSTM hidden layer with 25 neurons

- Ta et al. (2020) presents a model with 2 LSTM hidden layers with 512 and 256 neurons respectively.

- Hao & Gao (2020) presents a model with 3 LSTM hidden layers with 10, 20, 30 neurons respectively.

These three models contain many epochs: 1000 and 4000 in the first and second architecture. In the last model, the authors did not specify this parameter but set an early-stopping strategy (see section 2.14) with a patience equal to 80, and this suggests that the number of epochs has a rather large value.

Our model's task is the stock price daily predictions, but, considering that we used normalized price as the input, the output results were not the stock price in absolute terms. Hence,

we had to invert the normalization formula to obtain stock price predictions.

We want to obtain a low loss function value (Equation 2.6), which indicates a good model performance, by setting a reasonable number of epochs. As said before, the model is trained and optimized at each epoch. Hence its Loss function will decrease after a certain number of epochs, but we could enter the overfitting zone after a specific time (as illustrated in section 2.6 figure 2.10). We applied the two regularization strategies introduced in section 2.14 to determine the correct number of LSTM cells given our dataset. Specifically, the Early Stopping is based on the mean squared error loss function with patience ψ equal to 15 while the dropout is applied to the hidden layers with p=0,05. Specifically, we run various LSTM algorithms with different hidden layers and neurons to understand how does the model performance change in relation to these parameters. We started from the simple structure proposed by Fischer & Krauss (2017), and increased the neurons and layers to see the impact on predictions. Finally, we chose the best among these models considering their performance.

Concerning the performance evaluation, we used the loss function as introduced in section 2.4. We selected two different kinds of loss functions to understand the model performance during the training. The first was the classical Mean Squared Error (MSE) defined by equation 2.6 in section 2.4. This function was also used as the Loss metrics to optimize the model through backpropagation and early-stopping strategy. The second function used is the Mean Absolute Error or "MAE" defined by the following equation.

$$MAE = \frac{\sum_{i=1}^{T} |y_i - y_i^{'}|}{T} \tag{3.1}$$

This represents another common metric which analysed the average magnitude of the errors in a set of predictions, without considering their direction. The comparison between these metrics could provide more information on model performance.

Concerning the test set evaluation, we used the Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{\sum_{i=1}^{T} (y_i - y_i^{'})^2}{T}} \tag{3.2}$$

The RMSE metric emphasizes the large error (very bad predictions) than the small one; instead of MAE, where all the errors are treated equally. Concerning our task, we could expect small errors in predictions, but we want to avoid large errors because, from the investors' point of view, this could lead to a loss of his wealth.

We used RMSE to evaluate both training and test set forecasting. The value near to zero meaning that the model provides good estimations. The loss metrics evaluate the normalized data of training set, while the test set evaluation is based on unnormalized prices (predictions of daily prices). This means that the test set RMSE values are bigger than those of the training set.

We had set ADAM as an optimizer and Keras performs backpropagation implicitly with no need for a particular command. Given the LSTM memory cells complexity, we decided to build an architecture with a relatively low number of neurons and hidden layers. The table 3.1 summarizes the average performance across constituents of three principal architectures. The RMSE measures the deviation of test set predicted price without normalization from realized prices. The RMSE values are approximated to two decimal places.
The model performance varied in relation to neurons and hidden layers. The simplest architectures, one hidden layer with 25 or 50 LSTM neurons, had a high $RMSE$ value across the constituents if compared to the third model with two hidden layers with 50 neurons for each layer. The number of epochs in a model with a sequence of 90 days varied considerably in relation to the architecture set. Recall that we used an early-stopping strategy, which blocked the model training if there is no improvement after 15 epochs. The lower level of epochs in one hidden layer case is due to the fact that the Loss function (MSE) last improvement was at 35 and 65 epochs, respectively. Hence, we chose the third structure: two hidden layers with fifty neurons each.

Table 3.1: LSTM model performance comparison

Sequence length: 90 days			
Number of hidden layer	Number of neurons per layer	Epochs	RMSE test set
1	25	50	16.87
1	50	80	15.12
2	50	170	3.75
Sequence length: 240 days			
Number of hidden layer	Number of neurons per layers	Epochs	RMSE test set
1	25	130	9.87
1	50	150	9.13
2	50	140	3.59

We set a number of epochs equal to 200, which could be considered relatively low compared to similar works for stock price predictions. However, the early-stopping strategy blocked the training around 140 epochs (240 days) and 170 epoch (90 days) across the constituents of S&P500. There were only sporadic cases where the training executes all the 200 epochs. This was a good result in terms of performance; we have obtained a good value of RMSE after a reasonable number of epochs.

Table 3.2 summarizes all the hyperparameters used for our deep learning algorithm with Long-short term memory network.

Given all these elements, we can run our algorithm and obtain the predicted stock price. Figure 3.2 illustrates our algorithm predictions for daily adjusted-close prices of APPLE with the two kind of input sequences chosen. The predictions, in yellow, follow the realized price

Table 3.2: LSTM model hyperparametrization setup

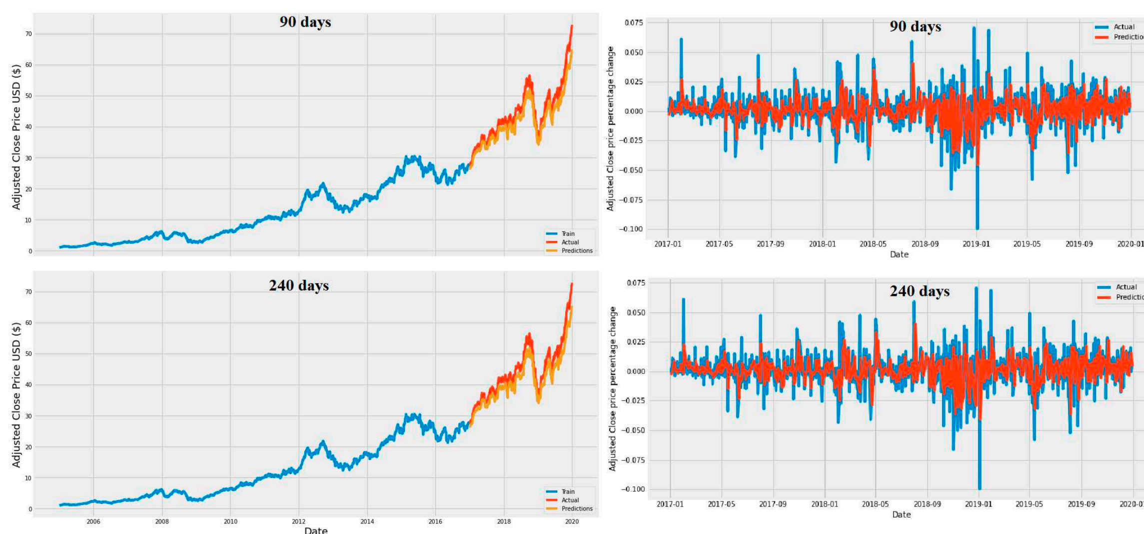
Categories	Hyperparameters
The number of hidden layers	2
Neurons of hidden layers	50
Neurons of output layer	1
Batch size	32
Number of epochs	200
Optimiser	Adam

movements (in red). However, we can see that the predicted prices are somehow lower than the realized prices.

The graph on the right represents the returns of stock prices obtained from the series of price prediction. The returns predicted by the model are in red. In general, the returns obtained from LSTM are lower in absolute terms and less volatile than the realized return.

The main difference between the two kinds of sequences regards the number of epochs. The

Figure 3.2: Daily adjusted stock price prediction: APPLE



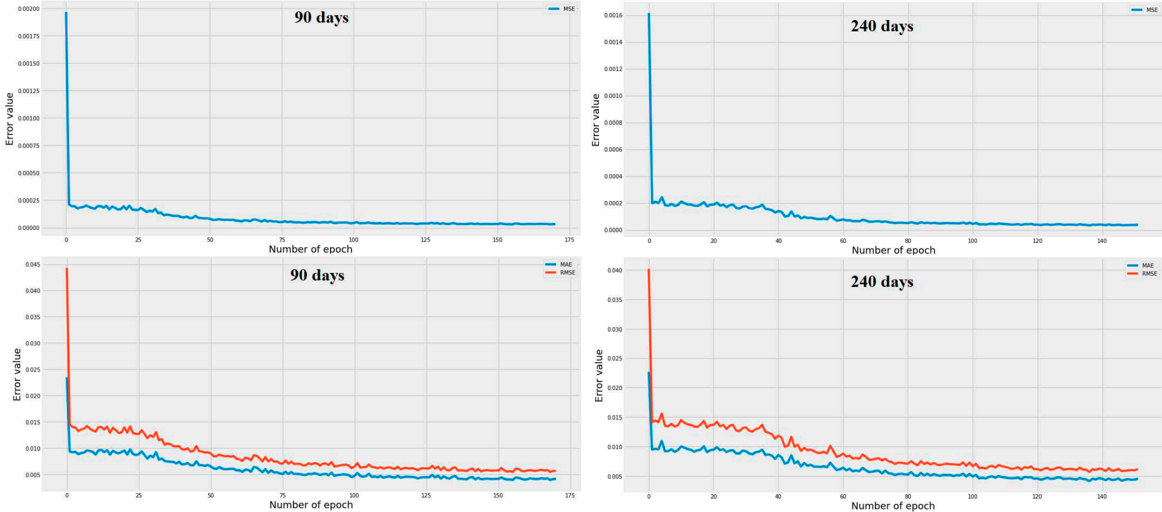
early-stopping blocked the training after 171 epochs in an algorithm with a sequence of 90 days while stopped the execution after 150 epochs in the other case. Figure 3.3 is useful to understand the loss metrics behavior which influenced this process. Recall that the early-stopping monitored specific metrics (MSE in our case) and stopped the training if there is no improvement in the model. The graphs at the top illustrate the MSE patterns across epochs. In both cases, the first attempts of training the model had relatively high error values, but after 40 epochs, error value started to decrease. After 100 epochs, it became low, but the small variations prevent the execution of early-stopping.

The other two loss metrics which followed a similar decreasing pattern across the epochs.

The loss metrics value across epochs followed this pattern across the various constituents of S&P500, hence it was not necessary to set a huge number of epochs as in Ta et al. (2020) and Hao & Gao (2020).

Table 3.3 summarizes the results of LSTM for Apple stocks with a comparison between the

Figure 3.3: Loss metrics on training set: a comparison



two kinds of sequences. We obtained a good result in terms of the stock's daily average return: the mean of daily predicted return is around 2% less than the realized return in both cases. The variance of daily prediction was less than 50% of the realized variance, as illustrated in Figure 3.2.

Table 3.3: Apple stock analysis with LSTM model: 01/2017-12/2019

Timestep	90 days	240 days
Mean of realized returns	0.001416	0.001416
Mean of predicted returns	0.001237091	0.001244016
Variance of realized returns	0.000241	0.000241
Variance of predicted returns	0.000100	0.000095
RMSE of test set	3.167366157	3.294366433

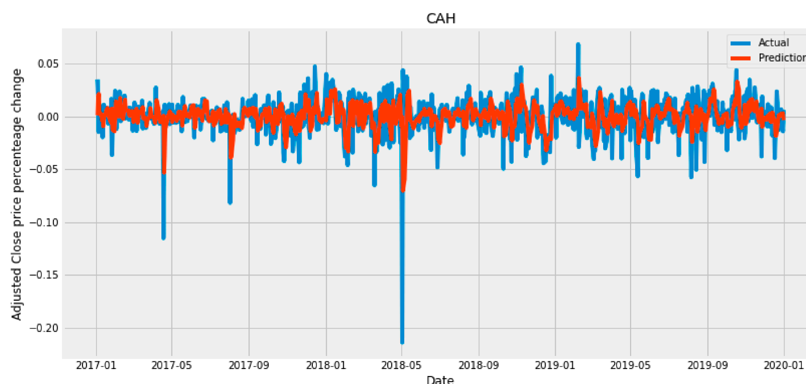
The results regarding average daily returns and RMSE were similar between the two kinds of input sequences, but the epochs value was lower with a sequence size of 240 days. For this reason, we decided to use this sequence for our algorithm. We processed the constituents with the LSTM architecture as illustrated in table 3.1 and collected the results for the daily adjusted stock price predictions to construct our investment strategy.

3.2 Algorithm performance across constituents

In the previous section, we illustrated the predictions of the LSTM algorithm concerning the Apple stock. However, the performance varied across the selected constituents of S&P 500 index. The early-stopping strategy blocked the model training at 130 epochs on average, but in some cases, the algorithm did not stop the execution until the end of 200 epochs. The loss function $RMSE$ value, which analyzed the test set error, is contained in a range between one and six. This indicated that there are some constituents where the LSTM model had lower or better performance than Apple.

Let us show some examples. We illustrated the daily stock returns obtained from price prediction because they are more explicative than absolute prices. Figure 3.4 illustrates the results for Cardinal Health Inc stock (CAH): the predictions follow the realized values in term of returns. However, the historical returns series contains some peaks that the model was not able to identify. These peaks can be due to various factors derived from macroeconomics sources,

Figure 3.4: CAH stock return predictions



industrial sector news, or company decisions. There are various examples: the company does not pursue the expected performance in terms of revenue or a new form of regulation that can impact a certain sector. Various stocks contain this peak identification problem.

However, the algorithm identified the patterns of stocks that are very volatile: see Figure 3.5, which illustrates the results for Oneok Inc. stocks. The historical returns exhibit a number of sizable variations across time, but we obtained good predictions.

We had various cases where our algorithm did not achieve a satisfactory performance. The prediction problems are related to the LSTM structure. Considering the Figure 2.14 in Section 2.8, the LSTM cell processed the input sequences with the relative time step x_t and controlled the flow of information with the various gates to obtain the prediction. Nevertheless, also the last predicted value h_{t-1} from the previously hidden layer is considered as a relevant indication to obtain this new output, as illustrated in Equation 2.29 of the output gate. The prediction obtained in timestep $t-1$ had a certain impact on the price prediction of the next timestep, and for this reason, it was difficult to detect the significant price change.

Figure 3.5: OKE stock return predictions

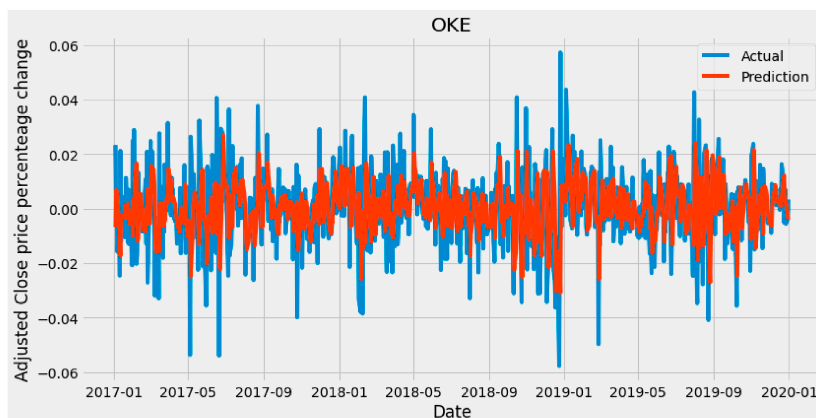


Figure 3.6 illustrates a case where the dependence of the h_{t-1} is exacerbated. The Figure shows the predicted returns of NOV inc corporations. We can see that predictions are close to realized return until January 2019. From this date onward, predictions start to deteriorate. The model was not able to identify the daily price variations. The realized returns became more volatile from 2019, indeed figure 3.6 shows how the historical series changed abruptly. This represented a relevant weakness of our predictive model.

Figure 3.6: NOV stock return predictions

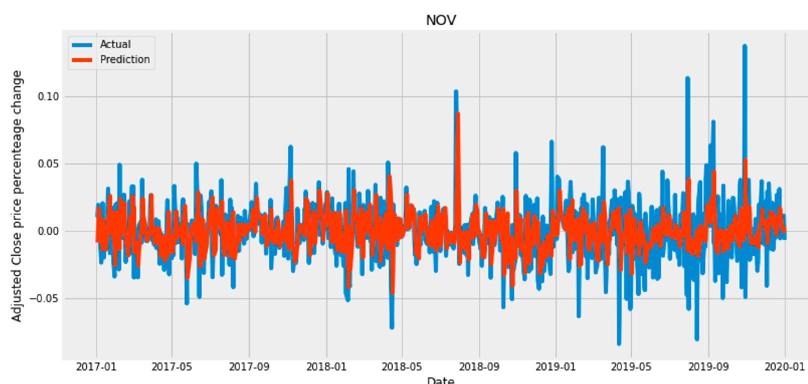


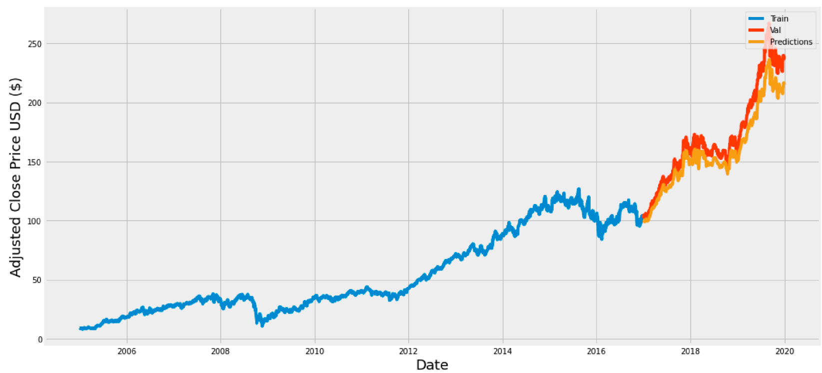
Table 3.1 indicates that the average test set loss function RMSE has an average value around 3 for our model. However, there are some outliers which must be considered. Figure 3.7 describes one of these cases where the model had a poor performance. The figure is referred to the SBA Communications Corporations (SBAC).
The algorithm executed 111 epochs with a RMSE loss metrics of 14. The plot of returns shows that the daily variations are bigger than the predictions. There are various case across constituents which exhibits this problem: specifically we have this bad performance for 44 constituents.

3.3 Accuracy of predictions

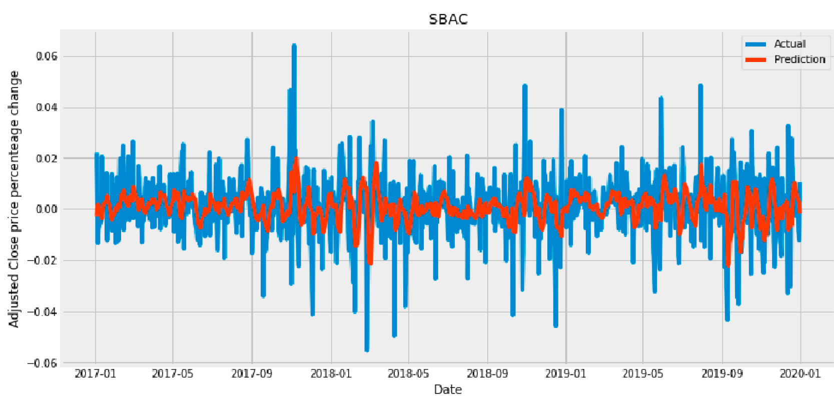
The task of this model is the estimation of daily stock price prediction. However, we want to evaluate if the model was able to identify the sign of daily return, because this is essential for

Figure 3.7: SBAC stock return predictions

(a) Adjusted closing price



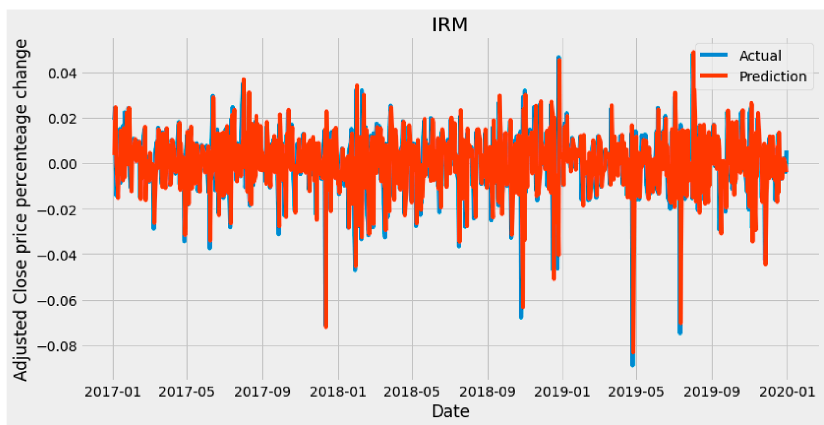
(b) Returns



the investment strategy. The accuracy indicates if the predicted returns are of the same sign (positive or negative) compare to the realized returns. Our LSTM structure does not provide a good performance in terms of accuracy, because there is a certain degree of heterogeneity across the constituents.

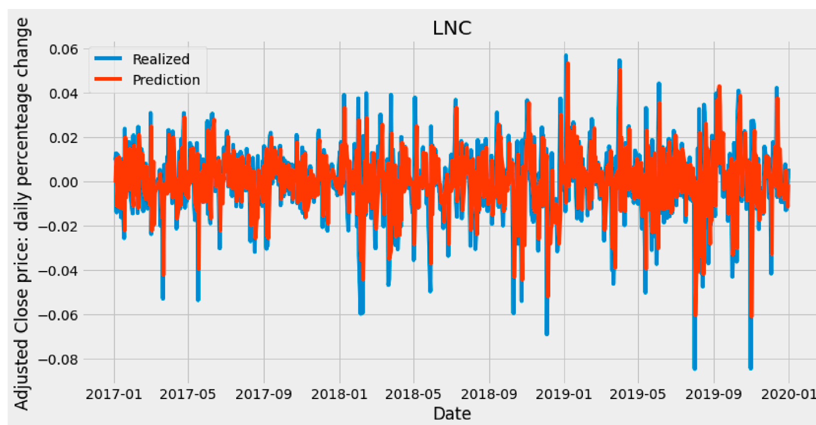
There are various price prediction which follow the realized price movements with a good accuracy. For example, the stock price predictions of Iron Mountain Inc (IRM) show a very good performance as illustrated in Figure 3.8. The accuracy is around 71%.

Figure 3.8: IRM stock return predictions



Another example is provided by Lincoln National Corporation (LNC) for which the accuracy is around 60%.

Figure 3.9: LNC stcok return predictions



This performance heterogeneity across the various constituents represents the main drawback of our model. In average we have an accuracy around 57%. The 44 constituents with very bad performance had an accuracy less than 40%, while we have 82 constituents for which the accuracy is around 70%.

We can evaluate these results by comparison of our model to the others proposed in literature, illustrated in section 3.1. The model provided by Fischer & Krauss (2017) tries to accurately predict whether a stock daily return outperforms the cross-sectional median of daily returns of all constituents or not. The accuracy is 51.4%.

The model provided by Hao & Gao (2020) identified a weekly trend of stock returns with an accuracy of 64%. However this model is not based only on LSTM architecture, but it is a combination of various layers based on different algorithm. Instead the model provided by Ta et al. (2020) does not give an indication about the accuracy of daily stock prices, and we cannot understand its predictive performance.

The accuracy metric provides a strong indication on these model capacity for stock price forecasting. We cannot consider these values as good accuracy and this means that the model has a low predictive capacity on average. However, some constitutes have a good performance with better accuracy (Figure 3.9 and Figure 3.8).

The value of accuracy represents a critical point of this model, this meaning that LSTM algorithm was not able to avoid the overfitting in average. These results are related to the high noise in the series of historical prices.

3.4 Investment strategy

The LSTM algorithm provided daily adjusted-close price predictions for three years (the test set): from January 2017 to December 2019. We build a daily rebalanced portfolio based on

these predictions. The short-term horizon of investment is a feature of quantitative trading, as illustrated in Chapter 1. Our portfolio construction strategy that we present is inspired by the study of Jegadeesh & Titman (1993): we decided to buy the winners stocks and selling losers. The winning stocks are stocks with the higher predicted daily returns, while the losers are the stocks with the lower predicted daily returns. Starting from the price predictions of the LSTM model, we constructed our daily portfolio as follows:

1. Collect the adjusted-close price prediction across the various S&P500 constituents.
2. Calculate the daily returns from price predictions.
3. Select the K stocks with the higher return for each trading day.
4. Select the Z stocks with the lower return for each trading day.
5. Enter in a daily long position on the K selected stock.
6. Enter in a daily short position on the Z selected stock.
7. Assign the same weights for each stock selected: equally balanced portfolio.

We set $K = Z = 10$; hence we selected the best (or winners) and worst (or losers) ten stocks. Each day we changed the stocks selected in our portfolio. We analyzed this strategy's results with and without transaction costs. In according to Avellaneda & Lee (2010) we set a transactions costs value of $5bps$.

In the previous section we illustrated the problem of accuracy across the constituents. We must consider this element for the investment strategy evaluation. The LSTM algorithm provides returns predictions which could be very different from the realized daily return of selected stocks.

This section is organized as follows: first, we present the results for the daily returns based on prevision. Then, we illustrate the results of daily return in realized terms. We decide to make this comparison to show the difference between the prediction and the realized price. Although the predictions seem to follow the realized price, as shown in the Figure 3.2 and Figure 3.7, the daily differences are relevant. After this comparison, we analyze our strategy's performance with annual metrics and risk indicators (VaR and CVaR illustrated in section 1.5). Finally, we compare the cumulative return derived from our strategy with the S&P 500 index to understand our investment performance properly.

After acquiring the daily return of selected stock, we calculated the daily return of this portfolio for each day. Table 3.4 illustrates the metrics of the daily returns with predicted values for the three years representing the test set, while the table 3.5 refers to the realized values of stocks. The mean return long indicates the mean of the daily return for the K stocks selected; the mean return short indicates the mean of daily returns for the Z stock. The mean return indicates the mean of returns of the entire portfolio, considering long and short position. The

other components refers to the daily returns calculated on entire portfolio for each year of test set. We selected the stocks with daily higher returns (long position) and lower daily returns

Table 3.4: Statistics metrics of predicted daily returns

Before transaction costs	2017	2018	2019
Mean return long(K)	0.027322	0.031547	0.032456
Mean return short(Z)	0.024775	0.031785	0.028604
Mean return	0.052097	0.063332	0.061060
Median	0.049815	0.060090	0.056512
Standard Deviation	0.013291	0.015506	0.016544
Sample Variance	0.000177	0.000240	0.000274

Table 3.5: Statistics metrics of realized daily returns

Before transaction costs	2017	2018	2019
Mean return long(K)	0.008308	-0.010594	-0.000043
Mean return short(Z)	-0.005095	-0.004675	-0.011525
Mean return	0.003213	-0.015269	-0.011568
Median	0.000826	-0.018091	-0.018794
Standard Deviation	0.115806	0.113463	0.125797
Sample Variance	0.013411	0.012874	0.015825
Kurtosis	3.594006	0.788857	2.623446
Skewness	-0.449880	0.028102	0.093859
Range	0.959622	0.739518	1.075146
Minimum	-0.559301	-0.412962	-0.613190
Maximum	0.400322	0.326556	0.461957

(short position). For these reasons, the means of daily predictions returns of our strategy are high in table 3.4: the daily return of 5% means that we obtained huge gains in one year. However, given the model accuracy around 57%, we often had realized daily returns that were lower or have a different sign. Many stocks selected for their high positive predicted return can have realized negative returns or the opposite.

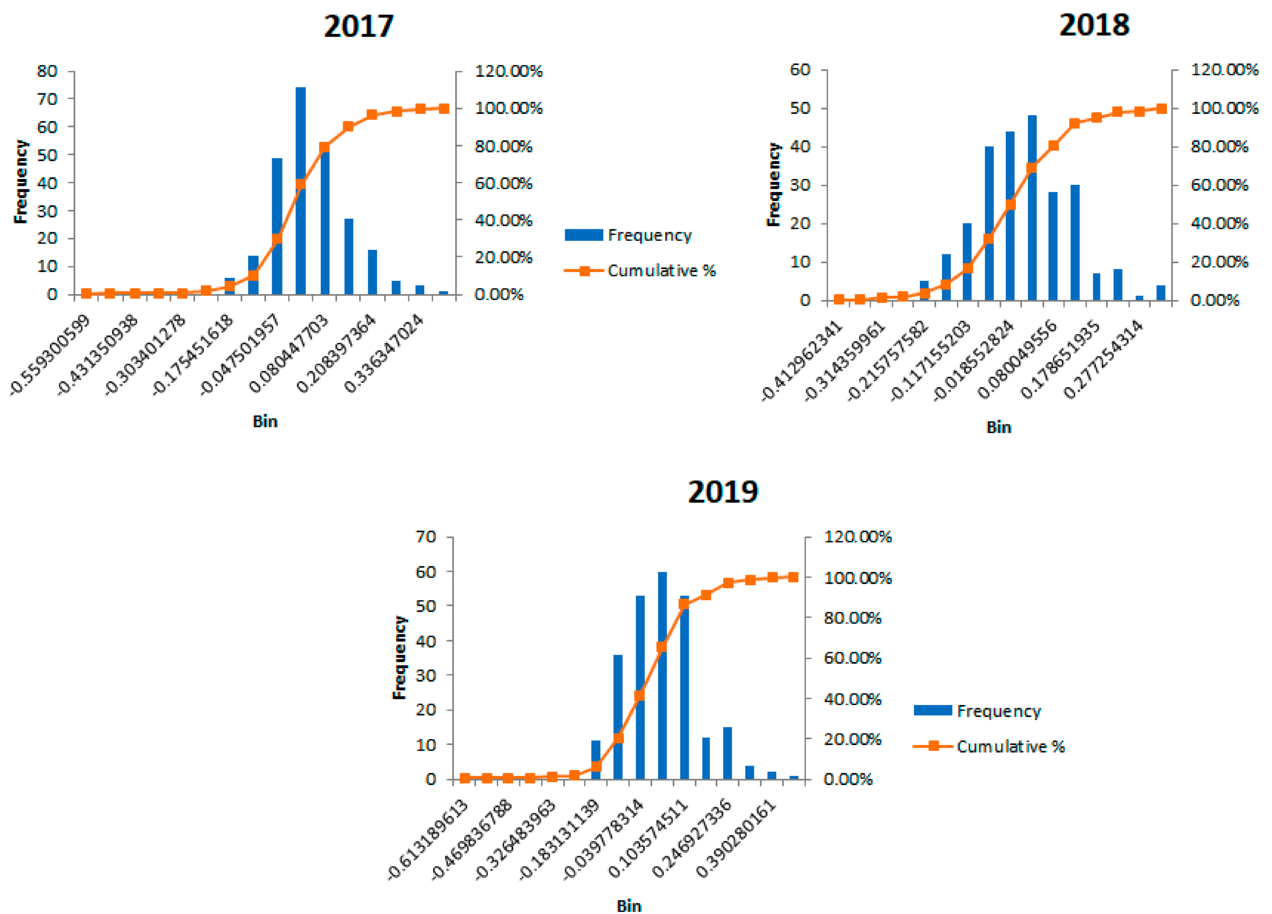
Considering these aspects, we can analyze our strategy's daily returns obtained with the realized data. As shown in table 3.5, the daily returns are positive on average only in the first year. Despite the presence of some stocks with good performance as explained in the previous section, the overall accuracy is not sufficient to obtain good results in terms of portfolio. The performance that we obtained is similar to Fischer & Krauss (2017), despite our period is different. The results in 2017 are due to the long position (K stocks); probably the selected stocks in this period included those with better predictions accuracy. The results are negative in other years (see section 3.5 for more details). The median is lower than the mean value,

and indicates a degree of asymmetry. This aspect, together with the maximum and minimum value, indicates the presence of some outliers that influence the daily mean values. Hence, our strategy could make considerable daily gains but also led to a daily loss.

The daily return across the test set years are asymmetric, as illustrated in the following histograms (Figure 3.10).In the first year of the test set, the daily returns are negatively skewed, and the Kurtosis value is greater than 3: the left tail is longer and fatter than a Gaussian. In 2018 and 2019, the distribution is positively skewed, and the Kurtosis value lower higher than 3: the right tails are thinner and shorter than Gaussian. These are indications of the presence of asymmetry.

We can note that the cumulative frequency of negative daily returns is around 30% of the total in 2017, and in a range between 40% and 60% in 2018 and 2019. Hence, our strategy based on LSTM prediction led to negative returns with a not negligible frequency. Specifically, our strategy has a very negative performance in 2018 and 2019 and this results in considerable losses. Table 3.6 summarizes the risk metrics VaR and CVaR of the daily return. The VaR

Figure 3.10: Daily portfolio returns histograms



represents the worst case loss scenario associated with a certain probability (section 1.5), in our case 1% and 5%. CVaR is derived by taking a weighted average of all losses in the tail of the distribution of possible returns, beyond the VaR threshold. Considering the daily av-

erage return values, respectively 0.003213, -0.015269 and -0.011568 across the years (as reported in table 3.5), the values of VaR and CVaR are relevant. Our portfolio strategy is associated with a certain degree of risk in 2017, and some days presented significant negative returns (losses). For the others two years the portfolio performance is negative, hence the risk metrics are higher.

Table 3.6: Risk metrics before transaction cost

VaR and CVaR analysis	2017	2018	2019
VaR 1%	-0.290331	-0.318290	-0.013443
CVaR 1%	-0.534109	-0.384772	-0.479177
VaR 5%	-0.168513	-0.186695	-0.192369
CVaR 5%	-0.270568	-0.257273	-0.270578

Table 3.10 summarizes the annual metrics: return, volatility and Sharpe ratio (see section 1.6). Following Fischer & Krauss (2017), we derived these annual metrics from the mean of daily return and standard deviation of the daily return. Given equation 1.8, it is necessary to determine the risk-free rate. We decided to use the yield rate of 10 years U.S. treasury bonds. The data are collected from the U.S Department of the Treasury website for the three test set years (2017, 2018, and 2019). We obtained a positive value in terms of the Sharpe ratio only 2018. In 2018 and 2019 the annual return and Sharpe ratio values are negative, following the results explained in the previous tables. However these results are similar to other works as Fischer & Krauss (2017) (see section 3.5). The results are not expressed as a percentage. There is another important element to consider: the transaction costs. The next tables sum-

Table 3.7: Annual metrics before transaction cost

Annual metrics	2017	2018	2019
Annual Return	2.225264	-0.996362	-0.985693
Annualized Standard Deviation	1.838371	1.801176	1.996969
Risk free rate	0.023295	0.029112	0.021393
Sharpe Ratio	1.197783	-0.569336	-0.504308

marizes the results of our portfolio strategies considering a level of transaction cost of 5bsp. This is a fixed cost that is applied to each selected stock.

We have the same shape of daily return histogram with respect to the case without transaction costs. There was only a small reduction of the various metrics. The VaR and CVaR, as well as the annual metrics, had slightly smaller values in this case.

Table 3.8: Statistics metrics of realized daily returns

After transaction costs	2017	2018	2019
Mean return long(K)	0.008267	-0.010541	-0.000043
Mean return short (Z)	-0.005069	-0.004652	-0.011467
Mean return	0.003197	-0.015193	-0.011510
Median	0.000821	-0.018001	-0.018700
Standard Deviation	0.115227	0.112896	0.125168
Sample Variance	0.013277	0.012746	0.015667
Kurtosis	3.594006	0.788857	2.623446
Skewness	-0.449880	0.028102	0.093859
Range	0.954824	0.735820	1.069770
Minimum	-0.556504	-0.410898	-0.610124
Maximum	0.398320	0.324923	0.459647

Table 3.9: Risk metrics after transaction cost

VaR and CVaR analysis	2017	2018	2019
VaR 1%	-0.288879	-0.316699	-0.327106
CVaR 1%	-0.531439	-0.382848	-0.476781
VaR 5%	-0.167671	-0.185762	-0.191407
CVaR 5%	-0.269216	-0.255986	-0.269225

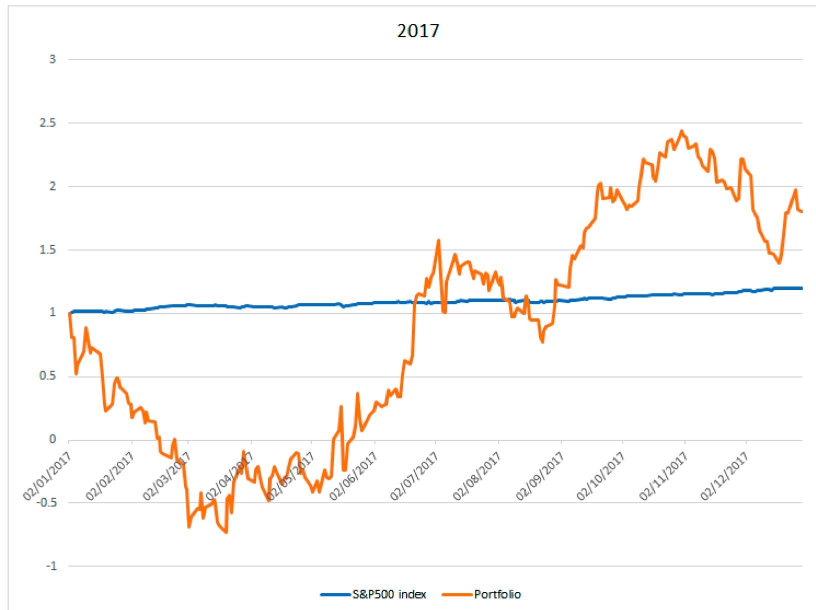
Table 3.10: Annual metrics after transaction cost

Annual metrics	2017	2018	2019
Annual Return	2.206465	-0.996257	-0.985384
Annualized Standard Deviation	1.829179	1.792170	1.986984
Risk free rate	0.023295	0.029112	0.021393
Sharpe Ratio	1.193525	-0.572139	-0.506686

3.5 Portfolio LSTM prediction based and S&P500 index: a comparison

Given the daily returns obtained from our strategy after transaction costs, we compared the yearly portfolio performance with the S&P 500 index (data collected from the website Yahoo finance). In particular, we analyzed the cumulative returns introduced in the section 1.4. We started from an initial wealth of one dollar to understand the evolution across the year.
The 2017 is the only year with a good performance. Figure 3.11 shows that our portfolio's cumulative return was lower than the S&P500 index in the first six months of the year, while the situation was the opposite in the second part of the year. The stocks selected with LSTM

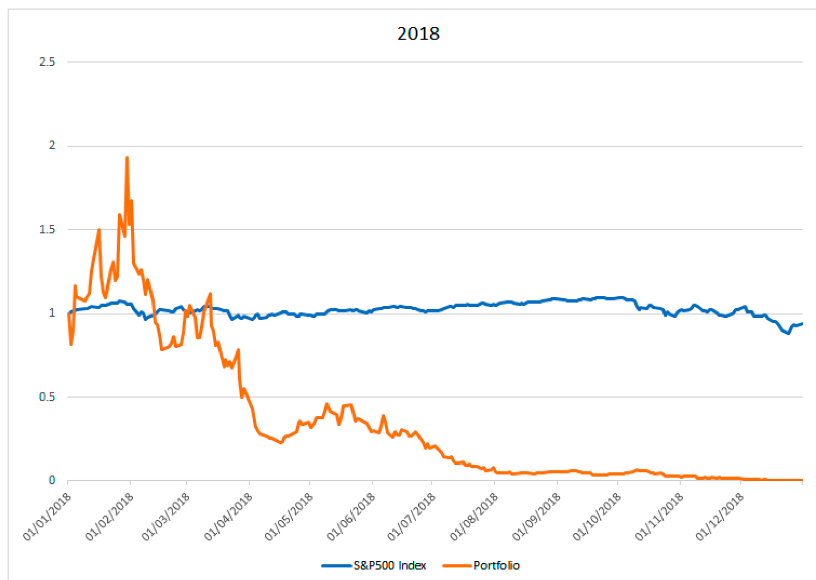
Figure 3.11: Cumulative return 2017



prediction show a negative performance, in particular in the first three months. However, after this period, the portfolio increased its performance in terms of cumulative return. The analysis of the portfolio performance is complicated because it does not follow the market index. This is due principally to the accuracy heterogeneity across various constituents. The stocks selected do not always have good prediction, this can lead to negative cumulative return. A possible explanation is that after march we selected stocks with higher model accuracy.

The second year of test set 2018 shows a different situation. Figure 3.12 illustrates the comparison of cumulative returns as in the case before.

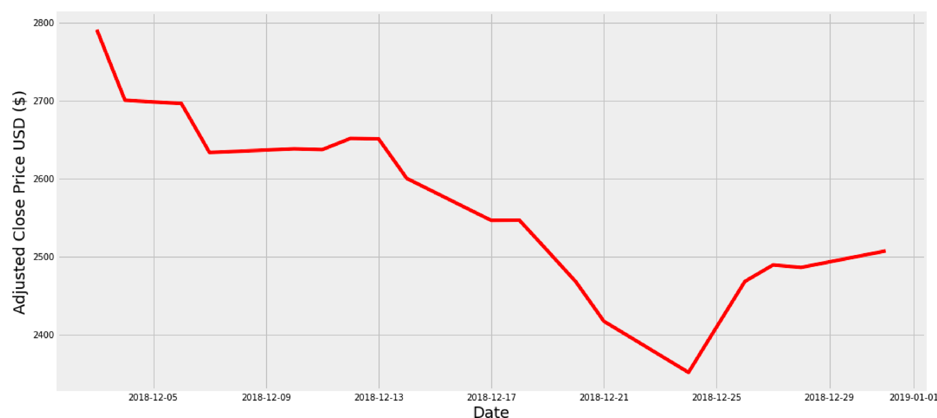
Figure 3.12: Cumulative return 2018



After April 2018 the model lead to relevant losses. The bad performance of S&P500 index

and the general bad predictions of daily returns across constituents resulted in a portfolio with negative Sharpe ratio. More stocks had negative returns; hence the probability of selecting these kinds of stocks was high. The S&P 500 index had a lousy performance in December 2018. See the Figure 3.13 which illustrates the December 2018 performance of S&P 500.

Figure 3.13: S&P500 index: Adjusted close price December 2018



There were three main events that lead to this performance (Rabinowitz & Shapiro, 2018):

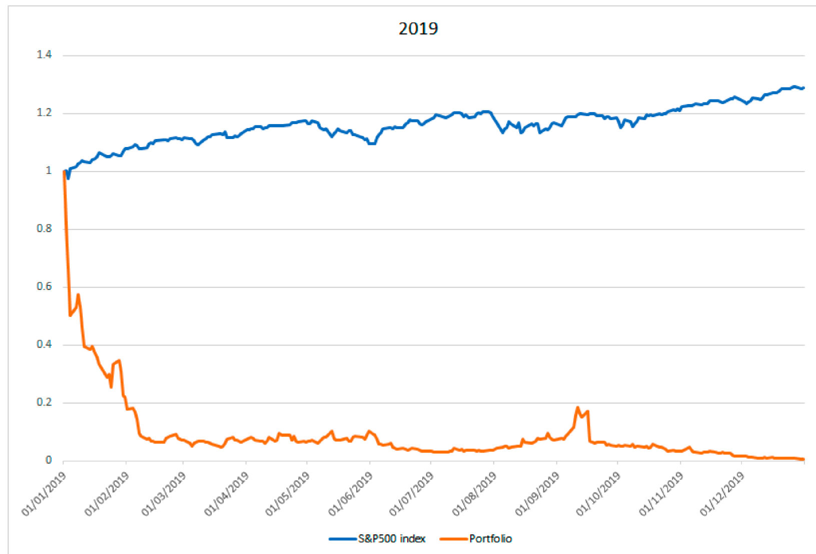
- 4 December 2018: Markets tumbled after Trump tweeted “I am a Tariff Man” and the Trump administration backed off earlier claims of a trade-war truce with China.
- 19 December 2018: The Federal Reserve announced the interest rate would increase from 2.25 percent to 2.5 percent, the fourth increase this year.
- 24 December 2018: The S&P 500 logged its worst Christmas Eve performance on record over a number of rising concerns, including the partial government shutdown; Trump’s repeated attacks on the Federal Reserve

The 2019 represents the last year of the test set of our model. Figure 3.14 illustrates the cumulative returns comparison. In this case, the portfolio presents a very negative performance. This means that the realized returns had the opposite sign with respect to the LSTM predictions. We presented in section 3.2 the drawback of this approach. However, we had worse results in this specific year than other periods of time of test set. Considering the figure 3.4, 3.7 and 3.6 in section 3.2, the predicted returns followed the realized values with less accuracy in the first eight months of 2019.

The LSTM makes daily prediction t from the previous 240 days, however as explained in section 3.2, the value of the previous predictions $t-1$ and the more recent data have a certain relevance. Hence, the bad performance of S&P500 index in the last months of 2018 could impact on these unsatisfactory predictions.

Given this situation, the Sharpe ratio of 2019 was lower than the other two years because we had higher volatility in the daily returns. This means that the impact of news and the performance of S&P500 in December 2018, made our prediction worse. This resulted in a

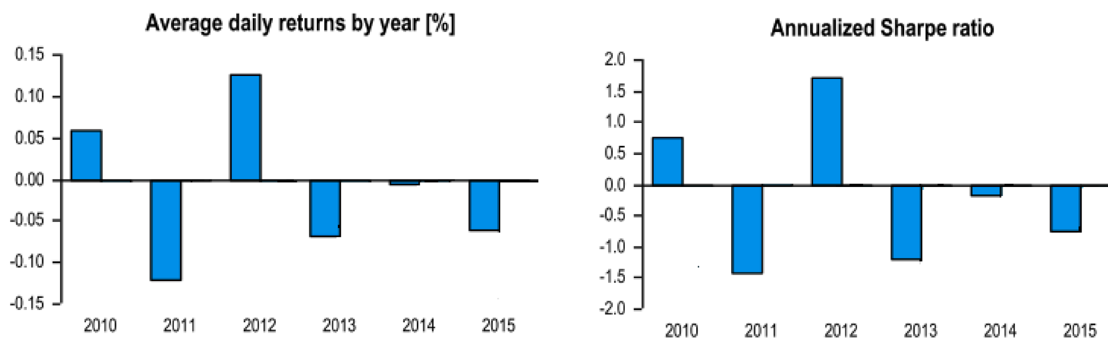
Figure 3.14: Cumulative return 2019



bad portfolio performance.

Our deep learning model does not provide a good investment strategy. However, the LSTM model with 1 hidden layer and 25 on S&P500 index provided by Fischer & Krauss (2017) from 2010 and 2015, presented a similar results as illustrated in Figure 3.15.

Figure 3.15: LSTM model by Fischer & Krauss (2017): results



Fischer & Krauss (2017)

The results present a negative average returns and Sharpe ration for four years, while the portfolio had a positive performance only in two years.

3.6 Model drawbacks

In section 3.5 we illustrated the cumulative returns of our portfolio strategy. The LSTM model predictions are based only to historical prices. However, this strategy did not consider various elements that could have a considerable impact on daily returns. Furthermore, our

algorithm structure was characterized by a distinct weakness. This section presents all these problems.

Algorithm architecture: Table 3.2 (in the section 3.1) illustrated the architecture of our LSTM algorithm with the relative number of hidden layer and neurons (LSTM cells). However, its performance varying across the various constituents (section 3.2). There were different elements that could influence the model performance:

- The number of observations of the dataset: we have collected 15 years daily stock price, but we can increase or decrease our input dataset's size.
- The length of input sequences
- The choice of regularization strategies
- The choice of hyperparameters

Finding the optimal value of all the elements to obtain a good performance of the algorithm across the constituents is challenging. There is the possibility that we can have some constituents with bad predictions.

Price movements accuracy: The algorithm could identify the price movement direction, in other words, if the return is positive or negative, with an average accuracy of 57% following related works. This had a relevant impact on our strategy. We selected stocks with a positive return prediction for long position and negative for short position, but we obtained lower profits because the realized returns were lower or had a different sign.

Understanding the overfitting: The identification of the overfitting problems can be challenging for a sequential dataset. We understood the evolution of loss metrics during the training. However, these metrics represented an average of the distance between prediction and realized value. If we have a relevant test dataset, as in our case (3 years), it is difficult to understand if the performance is good considering only the loss function for the entire dataset. As explained in the previous section, there is a possibility that the performance varies across time.

Time of the algorithm execution: Processing the input dataset required time. For a daily strategy, this is an element that cannot be underestimated. For example, our algorithm processed each stock in five minutes. This problem can be overcome with powerful hardware (e.g GPU with 32 or 64 Gb ram).

Slippage: The daily return were estimated on the assumption that we bought and sold the stocks at the adjusted-close price. However, in the real world, trading operations are more

complicated. Slippage is the term for when the price at which a trade order is executed does not match the price at which it was requested. One reason is the bid-ask spread, which represents the difference between ask and bid price. The bid represents the highest price that a buyer is willing to pay for the stock, and ask is the lowest price that seller is willing to accept.

This phenomenon can have a relevant impact on the portfolio performance. Our strategy was based on buying and selling stocks with a daily frequency, if the trade implied a lower or higher price than adjusted-close, the gain could become lower or could disappear.

Impact of unexpected news or event: The predictions were based only on historical data. However, many elements can lead investors to buy or sell a certain stock. In section we explained the performance of S&P 500 index performance in December 2018 given the decision of the Federal Reserve. Bubble, excess of optimism, events that increased the uncertainty in the future can affect investors' behavior, and a stock price that the deep learning algorithm cannot identify. Set an investment strategy considering only the adjusted-close price is too simple. There are a lot of elements that must be considered: stock volume, news of industrial sector or political events are some example.

Conclusion

In this work, we applied the Long-short term memory (LSTM) for stock price prediction on the constituents of S&P 500 index in a period from January 2005 until December 2019. The LSTM is a recurrent neural network architecture, which represents a specific model of machine learning, a particular subset of the Artificial Intelligence regarding building machines that could learn relationships between variables. We used this approach to predict the daily stock price from its historical series of past values. These financial time series daily predictions could be used to construct a specific portfolio daily rebalanced following the quantitative trading approach.

Specifically, the daily adjusted close stock price prediction considers the sequences of its previous 240 observations through the following function.

$$P_{t+1} = f_\theta(P_t, P_{t-1}, P_{t-2}, \cdots, P_{t-N+1})$$

The model must "learn" the value of the parameters θ. Our LSTM algorithm was built with a principal purpose: avoiding the overfitting problem. We adopted various strategies to create a non-complex model which was able to make predictions with a relatively small number of train attempts.

Deep learning models are notoriously difficult to train (Krauss et al. 2016), hence our model had some drawbacks. It cannot identified historical stock price peak and there were problems to identify the correct parameters of the forecasting function when the variance of time series changed across the years; hence the predictions had a considerable degree of error in some cases. This approach identified if the stock return was positive or negative with a degree of accuracy of 57% on average. The performance is higher for some constituents, 82 constituents presents an accuracy around 70%, and we have 44 constituents were the accuracy is very low, less than 40%. The accuracy of daily return cannot be considered satisfactory on average. Despite in various case we obtain good results, we are not able to overcome all the problems presented in other works based on similar approach as Fischer & Krauss (2017).

The price predictions obtained were collected to calculate the daily stock return in order to build a specific portfolio strategy: following the methodology proposed by various work as Fischer & Krauss (2017), we bought the daily best stocks in terms of return, and we shorted the worse stocks for the three years of the test set. We changed the stocks in our portfolio each day: a short-term investment representing the typical horizon of quantitative trading.

We obtained an average daily return after transaction costs of 0.31% in 2017, -1.51% 2018 and -1.15% in 2019. The Sharpe ratios had a value of 1.19 and -0.57 and -0.50 in annual

terms, respectively in 2017, 2018, and 2019. The LSTM algorithm cannot extract meaningful information from noisy financial time series data with a good performance across the various constituents, and the strategy based on this approach resulted in a profitable yearly portfolio in terms of cumulative returns only for the first year. Stock price predictions were good only in 2017, and probably we were able to select the constituents which are characterized from good accuracy. However, we had different periods where the daily returns are negative, this situation derived from the fact that in 2018 and 2019 the predictions become worse. The reason could be the drop of S&P500 index in December 2018 which had a considerable impact of predictions.

The LSTM algorithm and the deep learning model represented a new stock price prediction approach but involved different drawbacks. The unexpected events could impact on predictions accuracy, and there was a possibility that the model entered in an overfitting zone despite the strategies adopted. Furthermore, our prediction model did not consider many elements that could impact our profits: e.g slippage. The stock market has a certain degree of noise and could fluctuate as the result of macroeconomics factors or individuals' overreaction and underreaction to certain news.

Given the results presented in this work, the stock price prediction represents a challenging task for the machine learning algorithm. The model performance cannot be considered satisfactory, but these predictions are based only on historical prices. An algorithm must consider different elements: political policy, industrial development, company news, the volume of stock traded and historical price. This could represents the direction of future development of the machine learning approach for stock price prediction.

Bibliography

Avellaneda, M. & Lee, J.H., 2008. Statistical arbitrage in the US equities market. *Quantitative Finance*, vol. 10, no. 7, pp. 761–782.

Barberis, N., Shleifer, A. & Vishny, R., 1998. A Model of Investor Sentiment. *Journal of Financial Economics*, vol. 49, no. 3, pp. 307-343.

Bengio, Y., 2012. Practical recommendations for gradient-based training of deep architectures. *Neural Networks: Tricks of the Trade*, vol. 7700, pp 437-478.

Chollet, F., Falbel, D., Allaire, J.J., Tang, Y., Van Der Bijl, W., Studer, M. & Keydana, S., 2015. Keras. Available at: <https://github.com/fchollet/keras> [Date: 06/02/2021].

Daniel, K., Hirshleifer, D. & Subrahmanyam, A., 1998. Investor psychology and security market under- and overreactions. *The Journal of Finance,* vol. 53, no. 6, pp. 1839-1885.

Dixon, M., Klabjan, D. & Bang, J., 2015. Implementing Deep Neural Networks for Financial Market Prediction on the Intel Xeon Phi. *In: Proceedings of the 8th Workshop on High Performance Computational Finance,* pp. 1–6.

Donges, N., 2020. Gradient Descent: an introduction to 1 of machine learning's most popular algorithms. Available at:< https://builtin.com/data-science/gradient-descent> [Date:06/02/2021].

Edwards, W., 1968. *Conservatism in Human Information Processing*. Wiley, New York

Fama, E. F., 1965. The Behavior of Stock-Market Prices. *The Journal of Business*, vol. 38, no. 1, pp. 34–105.

Fama, E. F., 1970. Efficient capital markets: A review of theory and empirical work. *The Journal of Finance*, vol. 25, no. 2, pp. 383–417.

Fisher, T. & Krauss, C., 2017. Deep learning with long short-term memory networks for financial market predictions. *European Journal of Operational Research*, vol. 270, no. 2, pp. 654-669.

Gers, F., Schmidhuber, J. & Cummins, F., 2000. Learning to Forget: Continual Prediction with LSTM. *Neural computation*, vol. 12, no. 10, pp. 2451–2471.

Goodfellow, I.J., Bengio, Y. & Courville, A., 2016. *Deep Learning*. MIT Press, USA.

Graves, A., 2013. Generating Sequences with Recurrent Neural Networks. *Computing Research Repository (CoRR).*

Green, J., Hand, J. R. M. & Zhang, X. F., 2013. The supraview of return predictive signals. *Review of Accounting Studies*, vol. 18, no. 3, pp. 692–730.

Hamilton, J. D., 1994. *Time Series Analysis*. Princeton University Press, New York.

Hao, Y. & Gao, Q. 2020. Predicting the Trend of Stock Market Index Using the Hybrid Neural Network Based on Multiple Time Scale Feature Learning. *Applied Sciences*, vol. 10, no. 11.

Hochreiter, S. & Schmidhuber, J., 1997. Long Short-Term Memory. *Neural Compute*, vol 9, no. 8, pp. 1735–1780.

Jacobs, H., 2015. What explains the dynamics of 100 anomalies? *Journal of Banking & Finance*, vol. 57, pp. 65–85.

Jegadeesh, N., 1990. Evidence of predictable behavior of security returns. *The Journal of Finance,* vol. 45, no. 3, pp. 881-898.

Jegadeesh, N. & Titman, S., 1993. Returns to Buying Winners and Selling Losers: Implications for Stock Market Efficiency. *The Journal of Finance*, vol. 48, no. 1, pp. 65–91.

Kingma, D. & Ba, J., 2014. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations.*

Krauss, C., Do, X. & Huck, N., 2016. Deep neural networks, gradient-boosted trees, random forests: Statistical arbitrage on the S&P 500. *European Journal of Operational Research*, vol. 259, no. 2, pp. 689–702.

Le, X. H., Ho, H.V., Lee, G. & Jung, S., 2019. Application of Long Short-Term Memory (LSTM) Neural Network for Flood Forecasting. *Water*, vol. 11, no. 7.

LeCun,Y. & Cortes, C., 2010. MNIST handwritten digit database. Available at:< http://yann.lecun.com/exdb/mnist/> [Date:06/02/2021].

Lendasse, A., Bodt, E., Wertz, V. & Verleysen, M, 2000. Non-linear financial time series forecasting application to the Bel 20 stock market index. *European Journal of Economic and Social Systems*, vol. 14, no. 1, pp. 81–91.

Mitchell, T. M. 1997. *Machine Learning*. McGraw-Hill, New York.

Moulliet, D., Stolzenbach, J, Majonek, A. & Volker, T., 2016. The expansion of Robo-Advisory in Wealth Management. *Deloitte Berlin*, 2016.

Nwankpa, C., Ijomah, W., Gachagan, A. & Marshall. S., 2020. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *University of Strathclyde Glasgow*.

Ozbayoglu, A. M., Gudelek, U., M. & Sezer, O.B., 2020. Deep Learning for Financial Applications: A Survey. *Applied Soft Computing.*

Rabinowitz, K. & Shapiro, L., 2018. Stocks are down after a volatile year, but that’s not the whole picture. *The Washington Post*, 31 December 2018. Available at: <https://www.washingtonpost.com/graphics/2018/business/stock-market-crash-comparison/> [Date: 06/02/2021].

Racine Ly, R., Fousseini Traore, F. & Khadim Dia, K., 2021. Forecasting Commodity Prices Using Long Short-Term Memory Neural Networks. *ArXiv.*

Sak, H., Senior, A. & Beaufays, F., 2014. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH*, pp. 338-342.

Salman, S. & Liu, X., 2019. Overfitting Mechanism and Avoidance in Deep Neural Networks. *ArXiv*.

Singh, S., Srivastava, A., Mi, L., Caselli, R. J., Chen, K. , D., E.M. & Wang, Y., 2017. Deep Learning based Classification of FDG-PET Data for Alzheimers Disease Categories. *Proc SPIE Int Soc Opt Eng*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958.

Ta, V.D., Liu, C.M. & Tadesse, D.S, 2020. Portfolio Optimization-Based Stock Prediction Using Long-Short Term Memory Network in Quantitative Trading. *Applied Science,* vol. 10, no. 2.