LUISS

Course of

SUPERVISOR

CO-SUPERVISOR

CANDIDATE

Academic Year

Contents

Chapter 1

Introduction

The role of algorithms and machine learning has been constantly growing and shaping the financial industry in the latest decades.

The first hedge fund that showed its reliance in systematic strategies based on algorithms is Renaissance Technologies, founded in 1982 by the mathematician James Simons. He started this investing revolution with the secretive Medallion Fund, which has earned an estimated annualized return of 66%[1] since 1982.

Among the most famous and successful quantitative hedge funds there are Citadel, Two Sigma, D.E. Shaw, and AQR (Applied Quantitative Research). AQR, founded in 1998, grew its asset under management by 48% in 2017, and by 29% in 2018 to $90 billion. Now it has nearly $145 billion under management. Two Sigma, founded by D.E. Shaw alumni John Overdeck and David Siegel in 2001, grew its asset under management from $8 billion in 2011 to $60 billion in 2022.

This growth is parallel to the rise in the availability of data, which of course permitted the development of artificial intelligence and complex quantitative models.

In this work it is provided a satisfying introduction of machine learning, of ar-

[1] Barone, Emilio and Barone, Gaia, Unscrambling Codes: From Hieroglyphs to Market News (March 4, 2022). International Journal on Natural Language Computing (IJNLC) Vol.11, No.6, December 2022, Available at SSRN: https://ssrn.com/abstract=4049797 or http://dx.doi.org/10.2139/ssrn.4049797

tificial neural networks, and, more specifically, of recurrent neural networks, which are the ones used for sequential data. As a matter of fact, the aim of this work is to demonstrate the strong predicting power of models based on recurrent neural networks. The model used will be applied for predicting the VSTOXX EUR Index (V2TX), the index which tracks the volatility of the EURO STOXX 50 Index[2].

The first chapter explains how volatility is measured in finance, what the VSTOXX Index is and how it is calculated, and gives an introduction to the volatility forecasting problem.

The second chapter provides an overview of the main concepts of machine learning and of how artificial neural networks work in practice.

The third chapter introduces recurrent neural networks, and gives a clear idea about their functioning, by applying them to a particular prediction task. The prediction will be performed by a set of models, starting from a very naive approach and arriving to more complicated deep learning models.

The fourth and last chapter is focused on the main subject of this thesis: predicting the VSTOXX Index using a recurrent neural network model. In particular, the chapter will describe the steps taken to build the model and its predictive power.

[2]Barone, Emilio and Barone, Gaia, Tracking The EURO STOXX 50 (August 28, 2022). Available at SSRN: https://ssrn.com/abstract=4203380 or http://dx.doi.org/10.2139/ssrn.4203380

Chapter 2

Volatility

2.1 Historical Volatility

The historical volatility is a measure of uncertainty of returns of an asset. It is expressed as a percentage and it is calculated using the standard deviation of the logarithmic returns over a certain period of time. In other terms, it quantifies how much the returns deviate from their average in a given period. The formula is:

$$\sigma = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(R_t - \bar{R})^2} \tag{2.1}$$

Where:

n = number of observations

$R_t = \ln(\frac{S_t}{S_{t-1}})$

$\bar{R}$ = average of R_t for t = 1,..,n

S_t = price at time t

However, the previous formula would refer to the population variance, which would assume to have all the data point of the entire population. As a matter of fact, when measuring the historical volatility, it is used the square root of the sample variance,

expressed as follows:

$$\sigma_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(R_t - \bar{R})^2 \tag{2.2}$$

In statistical terms, the sample variance is an unbiased estimator and it is a snapshot from a larger, and unknown, population. The measure will be slightly higher to capture uncertainty.

For convention, in the derivatives environment historical volatility is often referred as realized volatility and it is calculated as an annual measure in percentage. Market participants consider the number of days in a trading year to be 252, and annual volatility can be converted in daily volatility using this simple formula:

$$\sigma_{day} = \frac{\sigma_{year}}{\sqrt{T}} \tag{2.3}$$

The usage of annualized volatility makes comparison very easy. For example, financial markets usually register volatility in a range between 15% and 20%, while volatility for stocks is usually higher and can reach levels higher than 100%. In this case it is possible to say that the stock in question, with a volatility of 100%, is five times more volatile than the market.

Regarding so, it is important to mention the beta of a stock. It is an additional measure to consider the risk of a security in relation to its benchmark. This value measures how much the value of the security changes, on average, for a one percentage point change in the value of the benchmark. The beta takes into consideration both correlation and risk and is a measure of co-movement, not volatility.

2.2 Implied Volatility

What has been described so far reflects past observations. However, a crucial aspect of financial options is implied volatility. In opposition to historical volatility, implied volatility is a forward-looking quantity and it is an indicator of the perceived future volatility of an underlying. Implied volatility is a metric derived from options prices and it is expressed annually and in percentage. It is then important to recall that the structure of an option premium is the sum of the intrinsic value and the extrinsic value. While the intrinsic value only depends on the strike price and the underlying price, the extrinsic value is influenced by many factors: maturity, dividends (if present), interest rate, and most importantly implied volatility. Intuitively, the more the implied volatility, the higher the premium of an option.

To calculate implied volatility there is not a specific formula but it is rather calculated in a way for which the theoretical price of an option is equal to its market price. The most common model used to find the theoretical price of European options for stocks that don't pay dividends is the Black-Scholes-Merton model, introduced in 1973 by Black and Scholes. According to the model, the price of the call is:

$$C = S_t N(d_1) - KN(d_2)e^{-rT} \tag{2.4}$$

$$d_1 = \frac{ln(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2})T}{\sigma\sqrt{T}} \tag{2.5}$$

$$d_2 = d_1 - \sigma\sqrt{T} \tag{2.6}$$

Where:

S_t = price of the stock at time t;

K = strike price

T = time to maturity (expressed in years terms)

r = continuously compounded risk-free rate

σ = historical volatility of the stock

$N(x)$ = cumulative distribution function of the standard normal distribution

The price of the put is instead expressed as following:

$$P = C - S_t - Ke^{-rT} \tag{2.7}$$

Which, replacing C and exploiting the normal distribution symmetry property, for which $1 - N(x) = N(-x)$, results in:

$$P = KN(-d_2)e^{-rT} - S_tN(-d_1) \tag{2.8}$$

However, from the formulas above it is not possible to calculate directly implied volatility, which is instead calculated by iterative methods. For example, given a certain call option price to be C, the following equation holds:

$$C = f(S_t, K, r, T, \sigma) \tag{2.9}$$

As the price of the call is observable in the market, there will be a certain volatility level for which the equation is respected.

The projected volatility of a stock throughout the course of the option's life is represented by implied volatility. Option premiums respond correctly when expectations alter. The market's estimate of the direction of the share price and the supply and demand of the underlying options have a direct impact on implied volatility. Therefore, implied volatility will grow as expectations or demand for an option rise.

Option premiums will be more expensive for options with higher implied volatility levels, and viceversa. This is significant because changes in implied volatility can

affect what the time value of the option will be, which can have an impact on the outcome of an option trade.

Every option responds to changes in implied volatility in a different way. For instance, longer-dated options will be more sensitive to implied volatility than shorter-dated options. This is because longer-dated options have higher time value priced into them, whereas shorter-dated options have lower time value.

Additionally, each option, depending on the strike price, will react to increases in implied volatility differently. Options with strike prices that are close to the money will be more sensitive to changes in implied volatility than options with strike prices that are more in-the-money or out-of-the-money. This is of course because options that are at-the-money are the ones with the highest extrinsic value.

2.2.1 Skew

Speaking of different implied volatilities for different strike price and same maturity, it is important to introduce the concept of skew. Volatility skew refers to the empirical relation between implied volatilities and strike prices for a given maturity.

As previously mentioned, implied volatility is retrieved from options prices observable in the market. These prices will fluctuate depending on demand and supply, and thus implied volatility will change accordingly. The skew gives an idea to investors about the demand for calls and puts.

Taking a step back, when the Black-Scholes model was introduced, the assumption was that options on the same underlying with the same maturity would have identical implied volatility. In reality, this supposition has been proved wrong. A standard explanation for the skew is that the return distribution, in opposition to one of the model's main assumptions, is not lognormal and is characterized by fat tails.

Volatility skew is usually referred as volatility smile, which is a long-observed

real-life pattern where at-the-money options tend to have lower implied volatility than in-the-money or out-of-the-money options.

After the so-called "Black Monday" (19^{th} October 1987), investors started to be more inclined to overpay for options that offered downside protection and, in the equity market, the volatility smile became a volatility smirk, as shown in Figure 2.1.

Figure 2.1. Volatility smile and volatility smirk.

Skew is very important for traders as it also gives an idea of the fear in the market. In a panicked market you would expect that investors selling options are probably afraid of selling more, so that supply will go down while demand will go up, resulting in an increase in options prices and thus implied volatility for certain options. In particular, the options that investors won't want to sell will be the deep-out-of-the-money ones or even the deep-in-the-money ones. Essentially, investors won't want to be involved in the tail risk. In these cases, not only it is possible to see implied volatility increase but also the fat tail nature of the return distribution, which invalidate the Black-Scholes lognormal assumption. So, not only traders will charge more for the increase in volatility for particular options but also for the increase in the tail risk.

Volatility skew is usually calculated as the difference in implied volatility between options with different strike prices but the same expiration date. However, volatility skew can be measured and quantified in several ways, thus there isn't a single method for determining it. Here are a few popular techniques for calculating volatility skew:

- Delta-neutral skew: this measures the difference in implied volatility between options with the same delta (i.e., the option's price sensitivity to changes in the underlying asset's price) but different strikes.
- Strike-neutral skew: this measures the difference in implied volatility between options with the same strike but different deltas.
- Absolute skew: this measures the difference in implied volatility between options with different strikes, regardless of their delta.
- Relative skew: this measures the difference in implied volatility between options with different strikes relative to their distance from the at-the-money (ATM) strike.

However, as suggested by Scott Mixon (2011), the most descriptive measure for volatility skew is given by this formula:

(25 delta put volatility – 25 delta call volatility) / (50 delta volatility)

For traders, volatility skew is crucial since it has an impact on the cost and risk of option strategies.

Traders can use volatility skew to their advantage by constructing option strategies that take advantage of the skew. For example, a trader might have a view on what the volatility skew would be in the market in the near future. In case of positive volatility skew, or forward skew, higher strikes correspond to higher implied volatilities. Thus, a trader could enter a call spread strategy buying call options with higher strikes while selling call options with lower strikes. On the other hand, a trader expecting negative volatility skew, or reverse skew, could open a put spread strategy buying put options with lower strikes while selling put options with higher strikes.

Volatility skew can therefore be used by traders as a tool for determining and evaluating market circumstances. A shift in volatility skew may signal a shift in

the sentiment of market participants or in the likelihood of large price swings in an underlying asset.
Like historical volatility, implied volatility is cyclical. Low-volatility periods follow high-volatility ones, and vice versa. Investors can choose the optimal trade by combining relative implied volatility ranges with forecasting methods.

Examining a chart is a useful strategy for determining implied volatility. Multiple implied volatility values are added up and averaged together on many charting systems in order to enable ways to chart many underlyings option's average implied volatility. For instance, market participants use indexes to monitor implied volatility. The most common index used is the CBOE Volatility Index (VIX). To calculate the VIX, the implied volatility values of near-dated, near-the-money S&P 500 index options are averaged. However, the VIX is calculated using American stocks. The index used to monitor implied volatility for European stocks is the VSTOXX Index, or V2TX.

2.3 The VSTOXX Index

To introduce the VSTOXX Index (V2TX) it is possible to use an expression from one of the most famous investors of all time, Warren Buffet: *"Be fearful when others are greedy. Be greedy when others are fearful"*. To give an idea of when investors are fearful or greedy two indexes are used as fear barometer: the American CBOE VIX Index and the European VSTOXX Index.
The focus of this work will be on the second index, which gauges the likelihood of movements over the next 30 days for the EURO STOXX 50 stock index, which tracks 50 Eurozone blue-chip stocks.
Figure 2.2 shows the VSTOXX Index from the beginning of January 1999 to the end of December 2022. Looking at the historical levels of the VSTOXX Index since

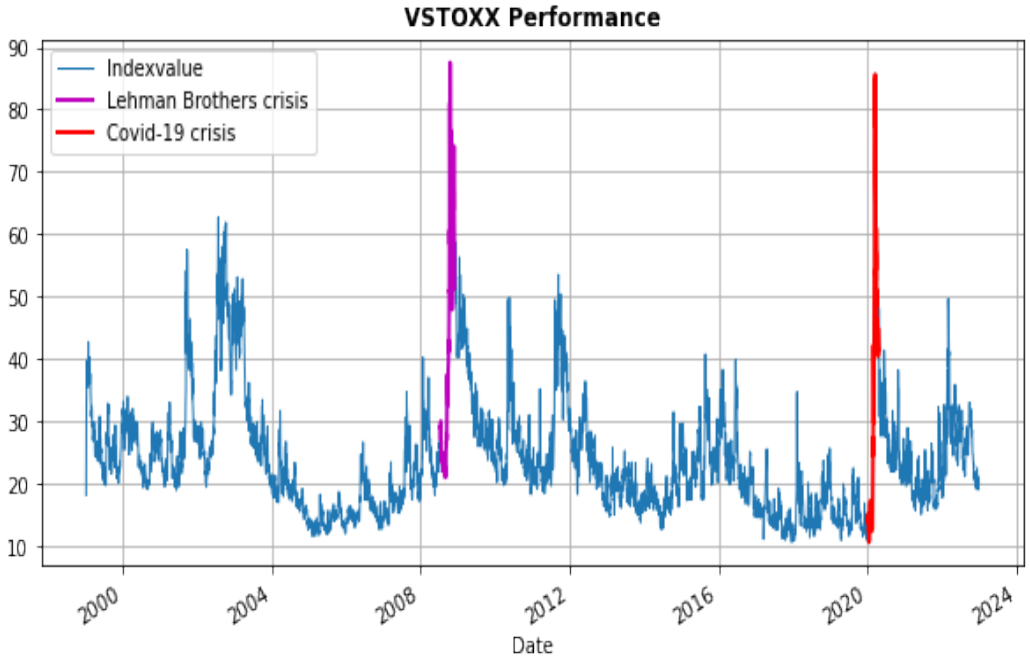

Figure 2.2. VSTOXX Index Performance from January 4^{th}, 1999 to December 31^{st}, 2022.

1999, it is possible to observe a pattern where predicted volatility "spikes" before gradually returning to a "normal" level, which is usually around 25. When the V2TX Index falls below 15, markets are considered to be “quite” or “greedy”, and the market index tends to climb gradually and steadily. On the other hand, markets are considered to be fearful when the index level is above 40.

The V2TX Index has only spiked above 80 twice so far, as highlighted in Figure 2.2, most recently in March 2020 when the COVID-19 pandemic began spreading exponentially in Europe, and in September 2008, when the Lehman Brothers bankruptcy sparked a global financial crisis. These two events caused the biggest and fastest shocks to the market.

There were also major surges above a level of 50 after the terrorist attack of September 11th, 2001, the failures of Enron and WorldCom in 2002, and the Greek debt crisis in 2011–2012. Despite the fact that volatility measures both directions of movement, the index tendency to spike during market downturns points to a "skew" in stock returns, where more than half of returns are positive and slightly above

average and are offset by relatively infrequent but sizable negative drops.

From figure 2.3 it is possible to capture the relation between the EURO STOXX 50 Index and the VSTOXX Index from 2007 to the end of December 2022.
It is interesting to note that each local maximum value of the VSTOXX Index matches a local minimum value of the EURO STOXX 50 Index. For example, the minimum level the EURO STOXX 50 registered was 2008, in correspondence with the maximum level for the VSTOXX Index.

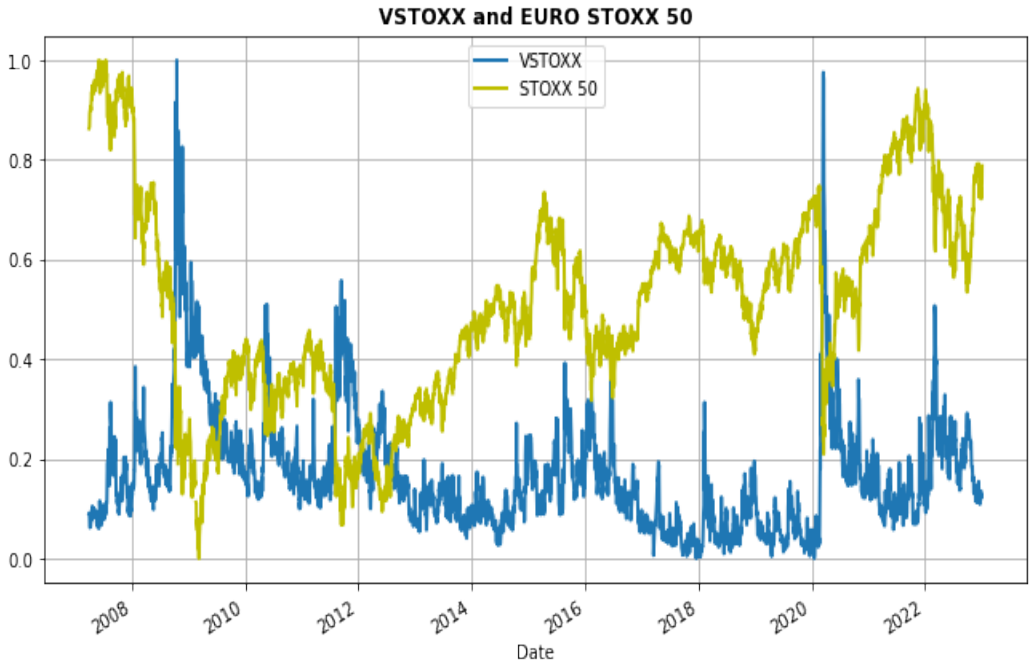

Figure 2.3. VSTOXX Index Performance versus EURO STOXX 50 Index from 2007 to 2022.

Even if the most used index is the one considering 30 days (VSTOXX), there are in total twelve main indexes for 30, 60, 90, 120, 150, 180, 210, 240, 270, 300, 330, and 360 days to expiry. As they are all calculated via interpolation, they are independent of a specific maturity and thus do not expiry, avoiding strong fluctuations upon

expiration.

In addition to the twelve main indexes there are 8 sub-indexes which cover EURO STOXX 50 option expiries from one month to two years. These sub-indexes are computed considering all the options available in the Eurex system.

The VSTOXX does not measure implied volatilities of at-the-money EURO STOXX 50 options, but the implied variance across all options of a given time to maturity. The model to calculate the EURO STOXX 50 Volatility Index has been jointly developed by Goldman Sachs and Deutsche Borse.

For the computation of each index tick, this is the input data:

- EURO STOXX 50 Index level
- OESX: best bid, best ask, last trade and settlement price of all EURO STOXX 50 options
- EONIA: Euro Overnight Index Average (overnight interest rate)
- EURIBOR: Euro Interbank Offered Rates
- REX: yield of the 2-year REX as the longer-term interest rate

As previously mentioned, the main indexes are calculated using a linear interpolation of sub-indexes whose expirations reflect the main index time to maturity. If one or both the sub-indexes necessary for the computation are not available, the main index is not calculated.

The formula used to calculate the main indexes is:

$$MainIndex_{tm} = 100\sqrt{\left[\frac{T_{st}}{T_{365}}\left(\frac{SubIndex_{st}}{100}\right)^2\frac{T_{lt}-T_{tm}}{T_{lt}-T_{st}} + \frac{T_{lt}}{T_{365}}\left(\frac{SubIndex_{lt}}{100}\right)^2\frac{T_{tm}-T_{st}}{T_{lt}-T_{st}}\right]\frac{T_{365}}{T_{tm}}} \tag{2.10}$$

Where:

- tm = time to maturity expressed as number of days

- SubIndex_{st} = VSTOXX sub-index with shorter maturity used in the interpolation

- SubIndex_{lt} = VSTOXX sub-index with longer maturity used in the interpolation

- T_{st} = seconds to expiry of SubIndex_{st}

- T_{lt} = seconds to expiry of SubIndex_{lt}

- T_{tm} = seconds to expiry of tm

- T_{365} = seconds in a standard year of 365 days (31,536,000)

The formula used to calculate the sub-indexes is:

$$SubIndex_i = 100\sqrt{\sigma_i^2}, \qquad i = 1, .., 8 \tag{2.11}$$

Where:

- σ_i^2= implied variance for the i^{th} OESX expiry date:

$$\sigma_i^2 = \frac{2}{T_i/T_{365}} \sum_j \frac{\Delta K_{i,j}}{K_{i,j}^2} R_i M_{K_{i,j}} - \frac{1}{T_i/T_{365}} \left(\frac{F_i}{K_{i,0}} - 1 \right)^2 \tag{2.12}$$

- T_i = seconds to the i^{th} OESX expiry date

- F_i = forward at-the-money price for the i^{th} OESX expiry date, derived from exercise price for which the absolute difference between call and put prices is smallest

- $\text{K}_{i,0}$ = highest exercise price not exceeding F_i

- $\text{K}_{i,j}$ = exercise price of the j^{th} out-of-the-money option

- $\Delta K_{i,j}$ = average distance between the exercise prices of the two options immediately above and immediately below $\text{K}_{i,j}$

- $\text{M}_{K\,i,j}$ = average of put and call prices of the option with exercise price of $\text{K}_{i,j}$

- $\text{R}_i = e^{r_i(T_i/T_{365})}$

- r_i = interpolated risk-free interest rate valid for the i^{th} OESX expiry date

2.4 Introduction to Volatility Forecasting

For the purpose of this work, it is important to introduce some of the main models used to forecast volatility. All the following methods are statistical methods commonly used for modeling, analyzing, and forecasting volatility in financial time series data.

The GARCH (Generalized Autoregressive Conditional Heteroskedasticity) model was developed on the basis of the ARCH (Autoregressive Conditional Heteroskedasticity) model, which was introduced by Robert Engle in 1982. The GARCH model is an extension of the ARCH model as it allows more flexibility in modeling the volatility of returns.

The basic form of a GARCH model is specified as GARCH(p,q). The first parameter, p, is the order of the autoregression component, and q is the order of the moving average component. The most common values used for p and q are (1,1). The model supposes that volatility comes in clusters, so that there are periods of high and low volatility, and that is why the GARCH model assumes that the volatility of returns is a function of both past returns and past volatility.
The model is defined by the following equation:

$$\sigma_t^2 = \omega + \sum_{i=1}^{p} \alpha_i r_{t-i}^2 + \sum_{i=1}^{q} \beta_i \sigma_{t-i}^2 \tag{2.13}$$

Where σ_t^2 is the conditional volatility of returns at time t, r_t is the return at time t, and α and β are parameters to be estimated.
The GARCH model forecasts future volatility by fitting the model to historical data,

and then using the estimated parameters to generate predictions for future volatility. The GARCH model has become a standard tool for volatility modeling, it's been widely used and well accepted in the finance and economics literature. However, like any model, it has its limitations and assumptions, for example, it assumes that the volatility is symmetric, and it assumes that the volatility is only influenced by past returns and volatility.

Unlike the GARCH model, the EWMA (Exponentially Weighted Moving Average) model does not require estimating many parameters, and it is a simple and efficient way to model volatility. The basic idea behind the EWMA model is that more recent observations are given more weight in the volatility estimate than older observations. This is done by assigning exponentially decreasing weights to the historical data. The formula for the EWMA model is:

$$\sigma_t^2 = \lambda\sigma_{t-1}^2 + (1-\lambda)r_t^2 \tag{2.14}$$

Where σ_t^2 is the conditional volatility at time t, r_t is the return at time t, λ is the smoothing constant between 0 and 1, and σ_{t-1}^2 is the previous volatility estimate. The lambda parameter determines the weight given to past observations; a higher lambda value means that more weight is given to more recent observations and less weight is given to older observations.
The EWMA model is computationally efficient and easy to implement, it also can easily accommodate for changing volatility over time. However, it does not consider the autocorrelation structure of the returns. Additionally, it may not capture the asymmetry of the volatility, which is common in financial data.

Finally, in an ARIMA (Autoregressive Integrated Moving Average) model, the volatility of the time series is modeled using a combination of differencing, autoregression, and moving average components. The differencing component accounts for any non-stationarity in the data, such as trends or seasonality, the autoregression component models the correlation between observations at different time lags, and the moving average component models the influence of past errors or shocks on the current observation.

The basic form of an ARIMA model is specified as ARIMA(p,d,q), where p is the order of the autoregression component, d is the degree of differencing, and q is the size of the moving average window.

It is important to note that, in order to use this model for volatility, the data should be transformed to returns, then the volatility can be estimated using the standard deviation of the returns.

Although the ARIMA model is widely used, it has some limitations. For example, it assumes that the volatility is constant over time, which may not be the case in real-world financial time series data.

2.5 Machine Learning for Volatility Forecasting

The use of machine learning to forecast the volatility of stock indices such as the S&P 500 and the EURO STOXX 50 is a relatively recent topic.

Researchers began experimenting neural networks and other machine learning techniques for volatility forecasting back to the late 1990s and early 2000s. However, rather than offering a thorough comparison with conventional volatility models, these early studies were primarily concerned with constructing models for volatility predictions.

With the advancements in machine learning algorithms and the availability of large

amounts of data in recent years, the use of machine learning for volatility forecasting has become more prevalent. For example, researchers have been using deep learning models such as recurrent neural networks (RNNs), long short-term memory (LSTMs), and other methods to forecast volatility of stock indexes such as S&P 500 and EURO STOXX 50. A lot of studies have been published in the recent years showing the improved performance of these machine learning models over traditional volatility models.

For example, Oliver Muncharaz, J. (2020)[3] published a work in which he makes predictions on the S&P 500 comparing classic time series models such as ARIMA against RNN models using LSTMs. The results he obtained show a significant reduction in prediction error when LSTMs are used.

Fischer, T. and Krauss, C.[4] published a work in 2017 showing the predicting power of RNN models. In this work, they aim at predicting out-of-sample directional movements for stocks of the S&P 500 form 1992 to 2015 using a LSTM neural network model.

Another interesting article was published by Hao, Y. and Gao, Q.[5] in 2020. They used a combination of convolutional neural networks and recurrent neural networks to predict price trend by analyzing the daily closing price dataset of the S&P 500 form 1999 to 2019.

Machine learning has significantly changed the way time series data is forecasted. It has introduced new techniques that can improve the accuracy of forecasts and handle complex data patterns.

[3]Oliver Muncharaz, J. (2020) Comparing classic time series models and the 137 LSTM recurrent neural network: An application to SP 500 stocks. *Finance, Markets and Valuation* 6(2), pp. 137–148.

[4]T. Fischer, C. Krauss, Deep learning with long short-term memory networks for financial market predictions, European Journal of Operational Research (2018), https://doi.org/10.1016/j.ejor.2017.11.054

[5]Y. Hao, Q. Gao, Predicting the Trend of Stock Market Index Using the Hybrid Neural Network Based on Multiple Time Scale Feature Learning (2020)

By incorporating neural network-based methods, machine learning has significantly altered time series forecasting. Recurrent neural networks (RNNs) and long short-term memory (LSTM) networks are two examples of those models that can manage enormous volume of data and recognize intricate patterns in it. They have been used to forecast time series for a variety of purposes, including stock price forecasting, traffic forecasting, and energy demand forecasting, and it has been demonstrated that they offer forecasts that are more precise than those made by conventional time series models.

In the next chapter, it will be provided an overview of how machine learning usage increased in the financial industry and of the functioning of neural networks.

Chapter 3

Basic Theory of Machine Learning

3.1 Machine Learning in Finance

Investors have long used previous data to forecast future results and based their choices on those results. As new tools and technologies were available, they progressed from simple univariate statistics to regression models, then more complicated models. Artificial intelligence and machine learning (ML) might be seen as the next step in this process.

Machine learning is a subfield of artificial intelligence: its goal, according to Samuel Arthur (1959), was *"to enable computers to learn from datasets without being explicitly programmed"*. ML seeks to develop algorithms that are effective at learning from sets of data and at producing predictions using statistical analysis methods.

Machine learning techniques, in their most basic form, are a collection of mathematical procedures that were created to allow computers to make predictions based on datasets of various types. Then, improve these predictions as the conditions become stable, and finally, adapt the results when faced with changing conditions. The goal of ML algorithms is then to identify the technique that accurately predicts the result

of the examined phenomenon on its own.

Algorithms used in machine learning extract rules or patterns from data in order to minimize prediction errors, for example. In order to decide whether to place buy or sell orders, traders must observe the market, analyze data, create predictions about the future, and manage the resulting portfolio to provide attractive returns relative to risk.

However, active investment management ultimately aims at providing alpha, which is defined as portfolio returns above the benchmark used for evaluation. The fundamental rule of active management states that having precise return estimates and the capacity to act on them is essential for producing alpha (Grinold 1989; Grinold and Kahn 2000).

The competition between investors on financial markets suggests that accurate forecasting to produce alpha necessitates superior knowledge, either through excellent data processing skills, an access to better data, or both.

The investing sector has changed significantly over the past few decades and has continued to do so to face cutting-edge technology, competition, and a difficult economic climate.

Among the trends that have contributed to the current prominence of machine learning and algorithmic trading there are the creation of investment plans that focus on risk-factor exposure rather than asset classes and the evolution of the market's microstructure, including the expansion of electronic trading and the blending of markets from various asset classes and regions. However, advancements in processing power, data collection and management, and statistical techniques including deep learning discoveries as well as the extraordinary performance of algorithmic trading pioneers in comparison to human investors played a crucial role.

Additionally, the 2001 and 2008 financial crises had a strong impact on the view

investors had on diversification and risk management. As a matter of fact, following the 2008 crisis, cost-conscious investors switched about $3.5 trillion from actively managed mutual funds to passively managed exchanged traded funds (ETFs).
Furthermore, as a results of increasing competition and different market conditions, hedge funds reduced their fees from the customary 2 percent annual management charge and 20 percent share of profits to an average of 1.48 percent and 17.4 percent, as per 2017.

Specifically, ML is used in trading as part of a specific strategy to achieve a certain corporate objective. Three waves have characterized the development and sophistication of quantitative strategies.
The first wave had been in the 1980s and 1990s, when most of the signals came from academic research and relied on just one or a small number of inputs generated from fundamentals and market data. These signals, which gave life to elementary strategies such as the mean-reversion strategy, are now fully commoditized and accessible through ETFs.
The second wave developed in the 2000s, due in large part to the groundbreaking work done by Eugene Fama, Kenneth French, and others, about factor-based investment. To look for arbitrage possibilities, funds used algorithms to find assets sensitive to risk factors such as value or momentum.
Finally, investments in machine learning capabilities and alternative data to produce profitable signals for repeatable trading methods are what propel the third wave.
An interesting consideration to take into account is that due to increasing market competition excess returns from new anomalies usually decrease by 25% from discovery to publishing and by over 50% after publication.

There are many objectives for which traders use algorithms in investment banks, hedge funds, and asset managements:

- to quickly price products for clients when involved in market making activity
- to execute trades at a favorable price
- to exploit arbitrage opportunities and profit from short-term trades
- to anticipate behavior of other market participants using behavioral strategies
- to use trading strategies based on predictions

Concerning the last point, the use of machine learning is quite relevant as with the use of neural networks it is possible to catch non-linear trends in large amounts of data. For example, in this work the focus will be on recurrent neural networks (RNN), the ones used for non-linear time series data. In order to understand RNN, it is important to provide a sufficient introduction to machine learning basic theory and then to artificial neural networks.

3.2 Machine Learning Overview

It was 1997 when Tom Mitchell provided the the following definition: *"a computer program learns from experience with respect to a task and a performance measure of whether the performance of the task improves with experience"*.
The main innovation machine learning brought is that the rules algorithms use to make decisions are learned from the dataset and are not programmed by humans. Experience is then presented in the form of training data.
There is a wide range of models to use depending on the data available and on the machine learning task. These models are grouped in four main categories:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning

- Reinforcement learning

The family of supervised learning models is the most used in the machine learning sphere. In supervised machine learning, a suggested by the name, the user gives the computer the data (or inputs) and examples of the intended outputs. The computer's task is to capture a general rule that links inputs to outputs, or, to put it another way, to identify an explanation for how outputs relate to inputs. The general rule captured is then used to catch the relationship in a set of new data.
There are significant trade-offs involved in learning an input-output connection that allows precise predictions of outcomes from new inputs. The most important trade-off is called bias-variance trade-off.
More sophisticated models using a range of variables are able to describe relationships with greater depth. However, they are also more likely to learn random noise belonging to a specific training sample. When this happens, the model is said to overfit to the training data. Furthermore, complicated models could be more challenging to examine, making it more challenging to comprehend the nature of the relationship or the forces behind particular predictions. On the other hand, more naive models could miss characteristics of the relation and fabricate biased results.

Classification and regression are the two main divisions of supervised learning. The output variable affects these two categories. For continuous output variables which respond to questions such as "how much?" or "how many?", regression methods are used. When the output variable is categorical (it answers questions with yes or no), classification methods are used. Examples of classification methods are random forest and support vector machines (SVM).

Instead, in unsupervised machine learning the algorithm has to find a pattern in the dataset without knowing a correct output. These algorithms identify hidden

patterns or data clusters without the assistance of a human. Because of their capacity to find similarities and differences in information, they are the best option for exploratory data analysis, cross-selling tactics, consumer segmentation, and picture identification.

Unsupervised algorithms seek to find structure in the input that enables a new representation of the information available in the data rather than making predictions about the future. As a result, it is usually not possible to compare the outcome to a ground truth as in the supervised scenario, and the quality may be subject to interpretation.

The main difference with supervised learning is that supervised machine learning contains both features (input variables) and labels (output variables or target), while unsupervised machine learning does not contain labels because of the different nature of the task.

The most common unsupervised learning techniques are clustering, association, and dimensionality reduction whose explanation is outside the scope of this work.

Semi-supervised learning is used for situation in which among a large set of data only a fraction of it is labeled. In this case, unsupervised learning methodologies are used to detect the structure of the features and supervised learning is used to create models to make predictions.

Finally, reinforcement learning is considered to be one of the most exciting and active area of machine learning. It is used for autonomous driving and for training robots while in the financing field can be used for dynamic option hedging, and portfolio management. It has a different approach than the supervised learning one as the model is based on a reward system where the model learns via trial and error. In reinforcement learning the model does not learn from examples but from experiencing the environment.

In extremely poor words, the reinforcement learning framework is composed by an environment that is operated by an agent which takes actions that change the environment and produce rewards. The scope is then to optimize the choice of actions to maximize the reward.

However, the main focus of this work will be on neural networks, whose models are part of the supervised machine learning ones.

3.3 Artificial Neural Networks

Artificial neural networks (ANNs) are a type of machine learning algorithm that are inspired by the structure and function of biological neurons in the human brain. As shown in the figure below, the first academic work concerning the subject came in 1943 from two neurophysiologists, McCulloch and Pitts, who wrote a paper describing how neurons might work and the structure of a simple neural network. Then, in 1958, Rosenblatt published the updated version of the paper in which he formally introduced a perceptron, which will be described better later in this work.

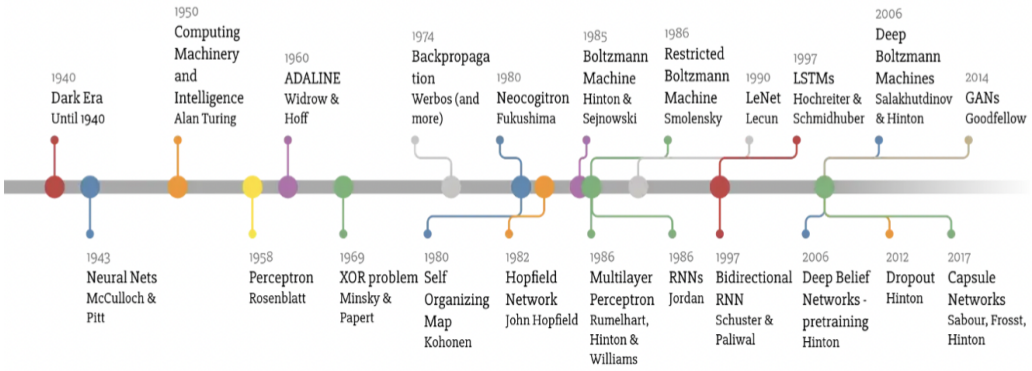

Figure 3.1. Machine learning timetable.

Similarly to biological neurons, artificial neurons in an ANN receive input, process it, and produce output. Usually, inputs and outputs of artificial neurons are numer-

ical values, and the computation performed on the input is a basic mathematical operation such as a dot product or a non-linear activation function.

Artificial neural networks can be considered as a simplified model of biological neurons. Of course, ANNs are not an exact replica of biological neurons. However, artificial neural networks are computational models that capture some of the key characteristics of biological neurons, such as the ability to receive input, process it, and transmit output.

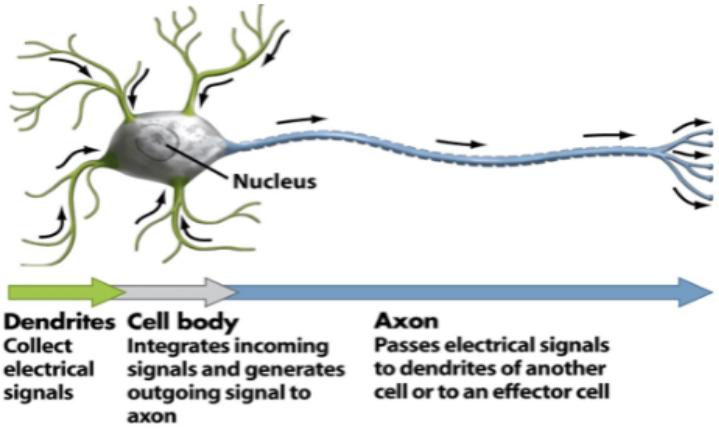

Figure 3.2. A simplified biological neuron.

Figure 3.2 helps to understand the functioning of a biological neuron.
A biological neuron is a cell that is specialized for transmitting information. In a nutshell, it receives input from other neurons or sensory cells through dendrites, processes this input through the cell body, and sends the processed information out through an axon. When a neuron is stimulated by input from other cells, it generates an electrical signal known as an action potential. This action potential travels down the axon and is transmitted to other neurons or target cells through synapses, which are the connection points between neurons and other nerve cells. The target cells can be other neurons, muscles, or gland cells. At the synapse, the action potential triggers the release of neurotransmitter molecules, which bind to receptors on the target cell and transmit the signal across the synapse. This transmission process is

how the brain and the nervous system communicate and process information.
The essential structure of a biological neuron is then composed by dendrites, cell body, axon, and synapses.

3.3.1 The Simple Perceptron

As cited before, the basics of neural networks models come from the simple perceptron. The simple perceptron was introduced by Rosenblatt in 1958 as a supervised binary classifier in which the input is represented by a vector of numbers.
The simple perceptron takes a vector of n input $\mathbf{x} = [x_0, .., x_n]$. Each of the vector components is assigned a weight (or strength) from a vector of weights $\mathbf{w} = [w_0, .., w_n]$. The weights are numerical values that control the level of importance of each input. The inputs are weighted and summed as follows:

$$z = \mathbf{w}^\mathbf{T}\mathbf{x} + b = b + w_1x_1 + w_2x_2 + ... + w_nx_n \tag{3.1}$$

Notice that the first term is a bias term that allows the algorithm to move the decision boundary from 0 in case all the features' values are zero. Once the vector signal is processed, an activation function converts it to get the final output, as shown by the figures below.

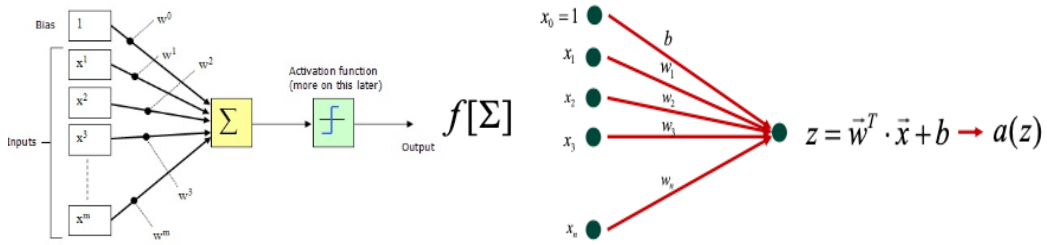

Figure 3.3. Representation of how a simple perceptron works.

The activation function in a simple perceptron is a step function:

$$a(\mathbf{w^T x} + b) = \begin{Bmatrix} 1 & if & \mathbf{w^T x} + b > 0 \\ 0 & if & otherwise \end{Bmatrix} \tag{3.2}$$

This is a discontinuous function whose derivative is undefined when the argument is zero. In neural networks there are other continuous activation functions that are used to perform this "transformation task".

However, the simple perceptron is a linear binary classifier with major limitations that restricted its applicability to real-life problems. For this reason, to solve these problems, hidden layers are added to the structure. A hidden layer is a layer which is neither an input nor an output layer.

3.3.2 The Multilayer Perceptron

The introduction of the multilayer perceptron (MLP) opens the door to a central subject of machine learning: deep learning (DL). With deep learning it is intended any artificial neural network with more than one hidden layer (multilayer neural networks). It is important to specify that MLPs are a specific type of multilayer neural networks (MLNN) that use a feedforward architecture. This means that the information flows through the network in only one direction, from the input layer through the hidden layers and to the output layer, without looping back. In contrast, a MLNN can have other types of architectures, such as recurrent architectures, in which the information can flow in multiple directions and loop back through the hidden layers.

Figure 3.4 shows the structure of a feedforward neural network with one hidden layer with three neurons.

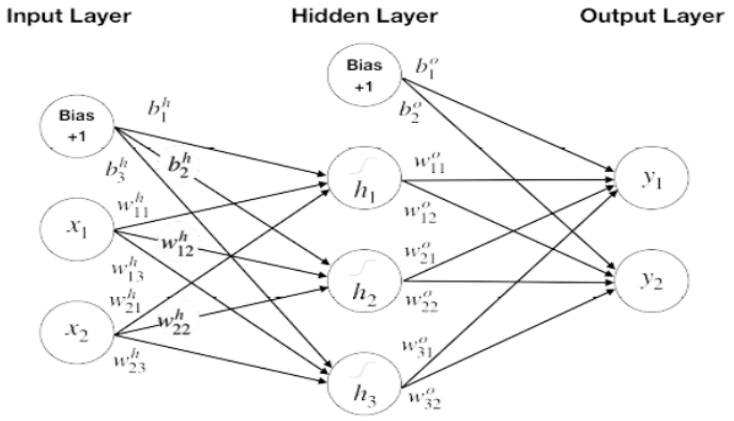

Figure 3.4. Feedforward neural network with one hidden layer.

Deep learning gets around many limitations that particularly restrict performance on high-dimensional and unstructured data that requires major efforts to extract informative features.

As already stated, the core challenge of supervised machine learning is to catch general rules for new samples using training data. As dimensionality of the data increases generalization becomes drastically more difficult.

Deep learning's founding principle is that the data was produced by a multi-level hierarchy of features. As a result, the assumption made by a DL model is that the target function is made up of a nested collection of simpler functions. Deep learning, in other terms, is a representation learning technique that draws a hierarchy of concepts from the data. It creates simple but non-linear functions that change the representation of one level (beginning with the input data) into a new representation at a higher, marginally more abstract level in order to learn this hierarchical structure. With enough of this transformations (and thus layers), deep learning can learn very complex functions.

The essential innovation is that deep learning algorithms are able to extract features suitable for modeling high-dimensional, unstructured data in a way that is infinitely more scalable than human engineering. Therefore, it is not surprising that the devel-

opment of deep learning coincides with the widespread access to huge amount of data.

The idea described before is stated by the universal approximation theorem, which formalizes the ability of neural networks to catch relationships between input data and output data. George Cybenko (1989) demonstrated that a neural network model with a single-layer using sigmoid activation functions can reproduce any continuous function on a closed and bounded subset of R^n. Kurt Hornik (1991) further showed that it is not the specific shape of the activation function but rather the multilayered architecture that enables the hierarchical feature representation, which in turn allows NNs to approximate universal functions.
The theorem does not, however, assist in determining the architecture of the network necessary to represent a given target function. The network's width and depth, the amount of connections between neurons, and the kind of activation functions are just a few of the many variables that need to be optimized.

Activation functions are a fundamental part of neural networks architectures. Without activation functions neural networks models would be linear regression models. This is because activation functions introduce non-linearity in order to capture and learn more complex relations between the data. Non-linearity is essential to make back-propagation, the process through which the algorithm modifies weights and biases, possible.

Let's consider a neural network with multiple layers l = 1,..,L:

$$\mathbf{a^{(1)}} = \sigma(\mathbf{w}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)})$$

$$\mathbf{a^{(2)}} = \sigma(\mathbf{w}^{(2)T}\mathbf{a}^{(1)} + \mathbf{b}^{(2)})$$

$$\mathbf{a^{(3)}} = \sigma(\mathbf{w}^{(3)T}\mathbf{a}^{(2)} + \mathbf{b}^{(3)})$$

$$...$$

$$\mathbf{a^{(L-1)}} = \sigma(\mathbf{w}^{(L-1)T}\mathbf{a}^{(L-2)} + \mathbf{b}^{(L-1)})$$

Where $\mathbf{a} = \sigma(\mathbf{z})$ and $\mathbf{z} = \mathbf{w}^T\mathbf{x}+\mathbf{b}$, considering a vectorized version for the hidden layers neurons. Each of these can be written as a vector operation. Operations in the first layer, in which j = 1,..,D neurons are considered, can be expressed as $a_j = \sigma(z_j)$ and $z_j = w^T\mathbf{x}+b$. Notice that the bias has a different value for each hidden layer neuron j.

The output of each activation functions decides whether a neuron is activated. If the input signal is high enough, the output from the activation function will be large and thus the neuron will be activated.

The main activation functions are shown below:

$$sigmoid: \quad f(z) = \frac{1}{1 + exp^{-z}} \tag{3.3}$$

$$tanh: \quad f(z) = \frac{exp^{z} - exp^{-z}}{exp^{z} + exp^{-z}} \tag{3.4}$$

The shape of the sigmoid and of the tangent hyperbolic function (tanh), as shown in Figure 3.5, is very similar. However, the tanh function has several desirable properties and it is often preferred to the sigmoid activation function. The first advantage of the tanh function is that it is centered around zero, meaning that the output of the function ranges from -1 to 1, included. This is useful because it

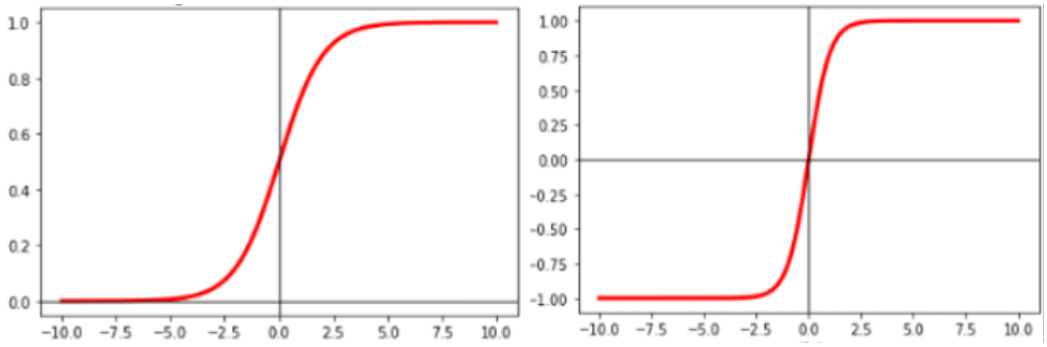

Figure 3.5. Representation the sigmoid function on the left and the tanh function on the right.

allows the network to model negative and positive values, which can help reduce the vanishing gradient problem during training.

The vanishing gradient problem is a phenomenon that occurs when the gradients of the parameters in the network become very small, something which can slow down the training process. It is often caused using activation functions that saturate for large input values, such as the sigmoid function. As a matter of fact, the output of the sigmoid function is always between 0 and 1, included, and the gradients of the parameters with respect to the output of the sigmoid function are always between 0 and 0.25, something that can obstruct the learning process of the network. In contrast, the output of the tanh function ranges from -1 to 1, and the gradients of the parameters with respect to the output of the tanh function are always between 0 and 1. In this case it easier for the network to learn.

For both of these activation functions when the input is very high or very low the gradient tends to zero. This results in a problem when training using backpropagation. For this problem the rectified linear unit (ReLU) activation function was introduced.

$$ReLU: \quad f(z) = max(0, z) \tag{3.5}$$

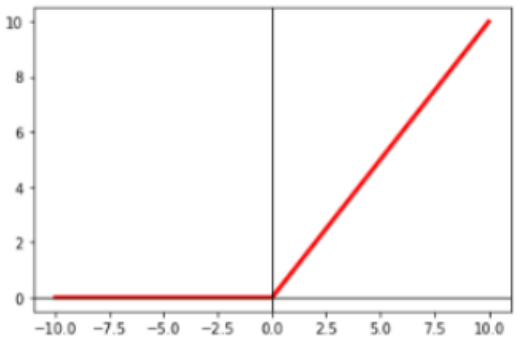

Figure 3.6. Representation the ReLU function.

The ReLU main advantage is that it helps to alleviate the vanishing gradient problem as it does not saturate for large input values, something which allows the gradients of the parameters to remain large and helps the network to learn more efficiently. Another advantage is that it is very simple to compute and is efficient to implement, which makes it easy to use in practice.
It has a derivative of zero for large negative inputs, which can lead to the problem of "dead neurons", but it is still better than the sigmoid and tanh.

For what concerns the output layer L, the activation function depends on the type of problem, which can be:

- Binary classification problems, which employ sigmoid activation functions at the output layer.
- Multi-class classification problems, which rely on soft-max, which produces a probability estimate for each class.
- Regression problems, for which instead the output layer is linear.

3.3.3 Training a Neural Network

It has been introduced before that neural networks are particularly interesting because of their capability to learn a desired pattern on their own, but how exactly do they do that?

The basic question is "what parameters should the model have in order to achieve such a result?". As introduced with the perceptron, the first move is to assign a weight for each one of the connections between a neuron and the neurons from the previous layer. A cost function is then needed to evaluate the performance of the model given a certain set of weights. This cost function is a measure of how much the predictions differ from the target value expressed by the label; thus, the goal of the training is to minimize the cost function.

Depending on the problem to solve the cost function varies. Considering the nomenclature used above, the regression cost function in given by:

$$C = \frac{1}{D}\sum_{j=1}^{D}(f(z_j^{(L)}) - y_j)^2 \tag{3.6}$$

Where D is the number of neurons in the last hidden layer.

For regression problems the output is continuous, and the cost function just illustrated is known as mean square error (MSE).

The next step is to calculate how sensitive the cost function is to a change in any of the networks' weight by computing the gradient of the loss function with respect to each weight.

In the case of the output layer L, where there is only one neuron in case of regressions, it is relatively easy to get the gradient.

The vector containing all the partial derivatives is called gradient and it is indicated

as follows:

$$\nabla f(\mathbf{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}\right]$$

In neural networks, in order to find the minimum of the cost function the iterative numerical method used is called gradient descent.

It is crucial to point out that gradient descent does not always work. As a matter of fact, all the functions to which gradient descent is applied must be differentiable. To recall, a function is differentiable when it is possible to calculate its derivative for each point of its domain.

Let's recall also that, intuitively, the gradient is the slope of the curve represented by a function at a given point; so, once the slope is known it is possible to determine the direction to move to.

Then, the gradient descent algorithm consists in calculating the derivative of the cost function with respect to each weight and then adjust the weights until the cost function is minimized, as illustrated below:

$$w_{ij}^{(l)} = w_{ij}^{(l)} - \eta \frac{\partial C(\mathbf{W})}{\partial w_{ij}^{(l)}} \tag{3.7}$$

The term $w_{ij}^{(l)}$ denotes the weight for the connection from the i^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.

The η term is called the learning rate, and it is used to control the size of the step. Regarding so, there isn't a precise learning rate to implement and it is often a matter of trial and error. This is because if η is too small convergence may be slow, and if it is too large, gradient descent may actually diverge.

The images in Figure 3.7 propose a visual idea of the process. On the left, there is a simplified example with only two dimension and one weight. As the slope is negative, with the gradient descent method the weight w will have to increase accordingly to the learning rate.

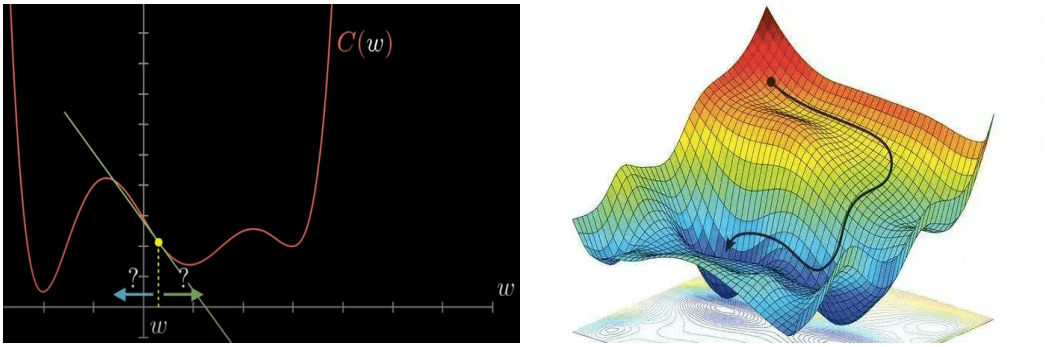

Figure 3.7. Graphical visualization of gradient descent.

The gradient descent method can be summarized in the following steps:

- Initialization in a starting point
- Calculate gradient of the point
- Make a step, whose magnitude is dependent on the learning rate, in the opposite direction of the gradient

The second and third point are then repeated until either a maximum number of iteration is reached or a step size is smaller than a pre-established threshold.

However, to calculate weights for previous layers there would be many derivatives to be computed. This is where the key concept of training ANNs come: backpropagation.

Backpropagation was introduced by Rumelhart, Hinton and Williams in 1986 and it is a method which exploits the chain rule of the gradient to calculate the partial derivatives with respect to all the weights in all the layers of a neural network simultaneously. It is called backpropagation because the algorithm starts by computing the error from the final layer and then it goes backward calculating the errors for the previous layers. Once all the errors for all the layers are calculated, the gradient descent method applies the chain rule to change all the weights and all the biases until the cost function is minimized.

On a final note, there are different aspects to consider when training a model. First of all, the model is trained over a number of epochs. An epoch is a hyperparameter that defines the number of times for which the algorithm goes through the entire training dataset. Typically, the number of epochs ranges from 10 to 100 even if, as cited before, the algorithm can stop earlier if no further improvement can be done to the model.

The initialization of weights is another important factor. Initially practitioners used $w_{ij} \sim N(0, 1)$.
Later it was shown that this initialization contributed to the vanishing/exploding gradient problem, and in 2010 by Gloriot and Bengio demonstrated that initializing the weights using the distribution proposed by Equation 3.8, gradient problems were significantly reduced.

$$w_{ij} \sim N\left(0, \frac{1}{\sqrt{0.5(N_{in} + N_{out})}}\right) \tag{3.8}$$

The term N_{in} refers to the number of incoming neurons in the layer and N_{out} the number of outcoming neurons.

Finally, the batch is the hyperparameters that controls the number of training samples to process before the model's parameters are updated.
For the gradient descent method (also called batch gradient descent), the batch size corresponds to the size of the whole training set. To simplify the process and to increase the speed of convergence, the stochastic gradient descent (SGD) was introduced. In this method the batch size is equal to one so that each parameter is updated after each and every sample.

However, there is also a solution in the middle of the two proposed above: the mini-batch gradient descent. In this case the batch size ranges from 2 to the size of the training set. This method is often the preferred because it increases the speed of convergence without significantly losing accuracy.

Chapter 4

Neural Networks Models for Sequential Data

4.1 Recurrent Neural Networks

In this part recurrent neural networks (RNNs) will be introduced. RNNs are a type of neural networks used to process sequential data where patterns evolve over time and where order matters.

In feedforward neural networks each sample is independent and identically distributed. However, there are some tasks for which the order of data matters. Among those there are speech recognition, translation, image captioning, and time series prediction, which will be the main focus later in this work.

The aim of recurrent neural network is to capture order-dependent information, and to do this, the algorithm requires a sort of memory for preceding data points. The most important innovation with RNNs, which had been introduced in 1986, is that the output is a function of both the previous output and the new input, and thus the information is incorporated from prior observations. The assumption of RNNs is that the input data gets generated as a sequence and that previous observations are relevant for predicting following values.

On the left of Figure 4.1, it is showed the basic structure of a feedforward neural

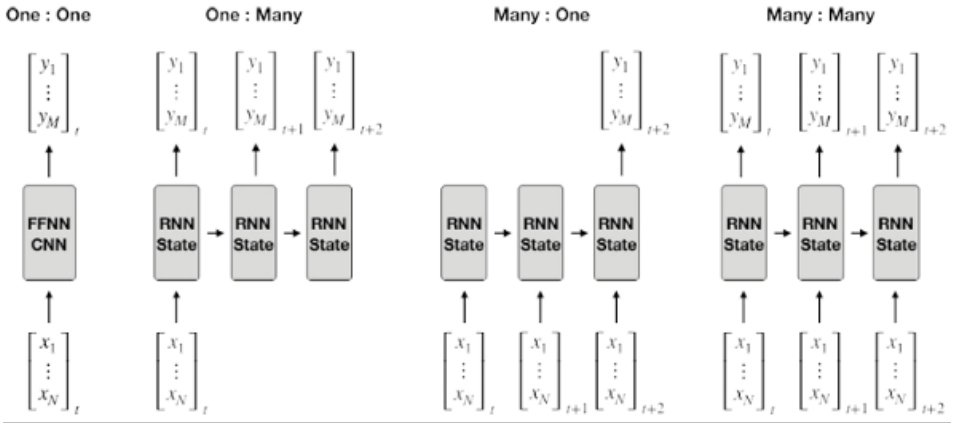

Figure 4.1. Main types of RNN models.

network. Here the action flowed in one direction; an input, usually a number or a vector of numbers, is aggregated to a set of weights and passed into an activation function to give an output.

The other three scenarios represent the main RNNs applications which include:

- One-to-many: it is used for image captioning. For example, a single vector of numbers (representing pixels) is the input, and the output is a sequence of words describing the image.
- Many-to-one: it is used for example in sentiment analysis. Each vector in the input represents a word and the output is a scalar or a vector.
- Many-to-many: it is used for translations or multi-step prediction of time series. These models will be applied later to real-life problems.

The functioning of a RNN is slightly different than the one previously analyzed.

In a recurrent neuron, the output is calculated by multiplying the inputs to the weights (and adding a bias). The same output then goes back in the network along with the next input to generate the next output. So, any time after time 0 there are two inputs in a RNN cell: the new input x_t and h_{t-1}. The last term is referred as

the hidden state and it is the term for which information from previous observations is preserved in the structure.

A single RNN cell can be described by the following equations:

$$h_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}) \tag{4.1}$$

$$\mathbf{y}_t = g(\mathbf{W}_{yh}\mathbf{h}_t) \tag{4.2}$$

Where $\mathbf{W}_{xh}$ is the matrix of weights connecting input with the hidden unit, $\mathbf{W}_{hh}$ is the matrix of weights connecting the previous hidden unit with the next hidden unit, and $\mathbf{W}_{yh}$ is the matrix of weights connecting the last hidden unit with the output. Usually, the activation functions used are tanh and ReLU. Figure 4.2 represents the functioning of a RNN by showing the unrolled computational graph of a single RNN cell:

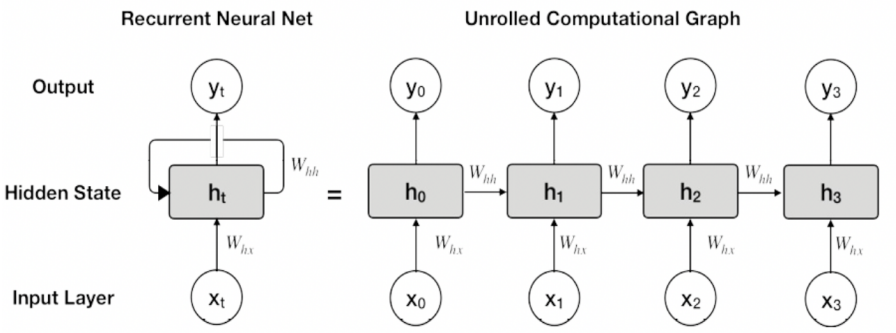

Figure 4.2. Computational graph of a single RNN cell.

This type of neural networks is called recurrent because in the model it applies the same operations to every element of a sequence so that the output will consider previous outcomes. For a better understanding of the idea behind it, RNNs can be seen as dynamical systems whose output changes over time. Referring to the state

of the system at time t as s^t:

$$s^{(t)} = f(s^{(t-1)}; \theta) \tag{4.3}$$

Intuitively, for a number of time steps equal to 3, the equation can be unfolded by applying the same definition 3 times:

$$s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta) \tag{4.4}$$

In the case of RNNs, the dynamical system also depends on an external signal x_t, which adds new information to the sequence:

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta) \tag{4.5}$$

Regarding training of recurrent neural networks models, a variation of backpropagation is used. The only difference, or complication, arising is that not only the error is propagated through layers, but also through time. As a matter of fact, the process is referred as backpropagation through time (BPTT).

4.2 Long Short-Term Memory

Although the training process does not have computational issues, it may cause the vanishing/exploding gradient problem when sequences in input get longer, something that may prevent the model to catch temporal dependencies.

To deal with this problem many approaches have been proposed. However, the most successful use gated units. With these units during the training process the model learns how much past information to retain in the current state and

how much information to reset. The most common examples are long-short term memory (LSTM) units, introduced in 1997 by Hochreiter and Schmidhuber, and gated recurrent units (GRU).

LSTMs have different gates which determines whether to keep an information and, if so, how much of that information to keep in the system.

From "outside" the LSTM cell looks the same as a RNN cell, however, the LSTM cell is far more complicated internally.

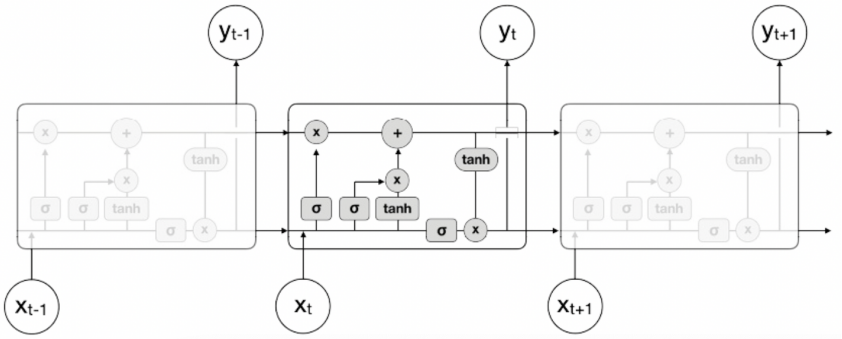

Figure 4.3. Inside the LSTM unit.

Figure 4.3 illustrates what a single LSTM cell looks like. Commonly, transforming and passing along vectors, a unit combines four parameterized layers that interact with each other. These layers are divided in input gate, output gate, and forget gate. Figure 4.4 helps to describe the functioning of the cell.

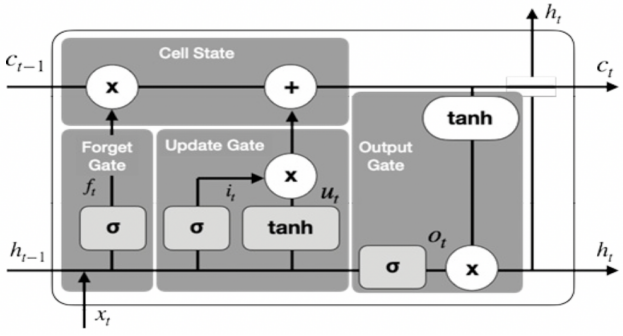

Figure 4.4. Functioning of a LSTM unit.

In the figure, the dark grey elements represent layers in which weights and biases are learned during training while the white circles symbolize element-wise operations.

The cell state, c, is the one on top of the cell and it has connections with all the other gates.

First, the forget gate controls how much information should get in the cell state. It receives two inputs: the hidden state h_{t-1}, and the current input x_t; it combines them and applies a sigmoid activation function:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{4.6}$$

The result will be in a range between 0 and 1, included, and will be multiplied by the cell state c_{t-1}, updating it accordingly.

Then, the update gate receives h_{t-1} and x_t as well and computes a sigmoid and a tanh on them. The two results, u_t and i_t are multiplied and then added or subtracted, depending on the sign, from the cell state.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{4.7}$$

$$u_t = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4.8}$$

Finally, in the output gate h_{t-1} and x_t go through a sigmoid function to output o_t.

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{4.9}$$

At the same time o_t is combined with the ultimate cell state c_t, which is normalized by the tanh function.

The final outputs of the unit are then h_t, which can be thought as the short-term memory, and c_t, which can be thought as the long-term memory.

$$c_t = f_t \odot c_{t-1} + i_t \odot u_t \tag{4.10}$$

$$h_t = o_t \odot tanh(c_t) \tag{4.11}$$

A simplification of LSTMs is provided by the GRUs (gated recurrent units), which were introduced in 2014 by Cho et al. These units combine the input gate and the forget gate into a single update gate. Even if they propose a simpler version of LSTMs, GRUs have been shown to achieve comparable performance especially on language tasks.

In order to properly introduce the core subject of this thesis, forecasting volatility, in the next subsection it will be illustrated a sequence of models whose aim is to predict future values of a time series. This subsection is exclusively a visual representation of the forecasting problem, and the aim is to provide the reader a deeper understanding of the reasoning for which RNN are the best choice for time series forecasting.

4.3 Predicting a Noisy Sine Wave

The first step is to generate the noisy sine wave. To be more precise, it was shown that in order to train a neural network model it is necessary to have a sufficient amount of data, so a set of noisy sine waves will be created.
Neural networks require a three-dimensional input, so the matrix will have size batch_size x n_steps x input_dimension, where batch_size refers to the number of time series samples, n_steps refers to the length of the sequence for each sample, and input_dimension the number of features used in the dataset.
The matrix is composed by 10000 samples, each representing a time series of 51 time steps. The 51^{st} element is the output, y, which the model wants to predict.
In machine learning, in order for the algorithm to be properly evaluated, the dataset is split in:

- Training set: the part of the dataset used to fit the model
- Validation set: the part used to tune hyperparameters (not compulsory)

- Test set: the part used for the final evaluation of the model

It has been shown that models have the best performance when the training set includes 70/80% of the dataset. Considering this, the matrix is split into training, validation, and test set which have respectively 7000, 2000, and 1000 samples.
Each set is composed by X and y. X has the first 50 elements of the sequence and y the last one, the one to be predicted. Figure 4.5 illustrates three random samples:

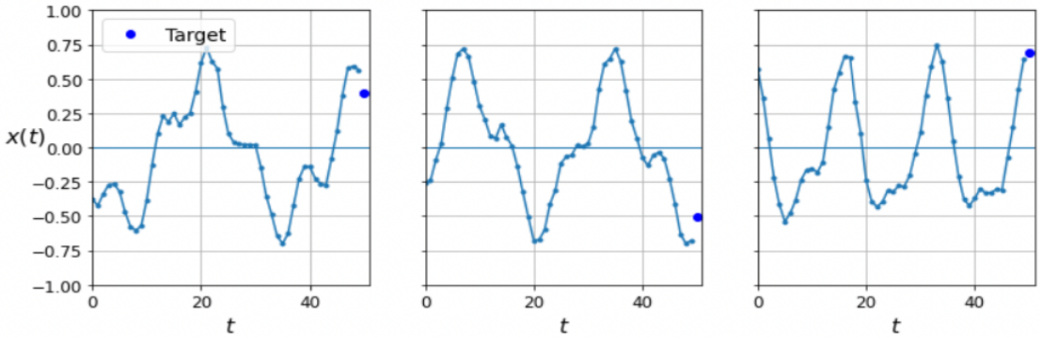

Figure 4.5. Three examples of noisy sine wave.

The blue dot represents the target value to be predicted.
The MSE will be used to evaluate the performance of the models.
At the beginning, all the models will aim at predicting the 51^{st} value using all the previous 50 values. To provide a better idea of the performance of the models, the same sample time series is considered.
The first approach, shown in Figure 4.6, is the most naive, and it uses the last value of X as a prediction of y. Again, the blue dot in the graph is the target output while the red cross is the estimate of the approach. The MSE for this attempt is equal to 0.02.

The second approach uses a basic feedforward neural network. The model has an input layer with 50 neurons, two hidden layers with 50 neurons each and a linear output layer with one neuron, as there is one target value to be predicted.
The model has 5151 parameters to be trained. Each neuron is connected with all the neurons of the following layer and each neuron has a bias to take into account.

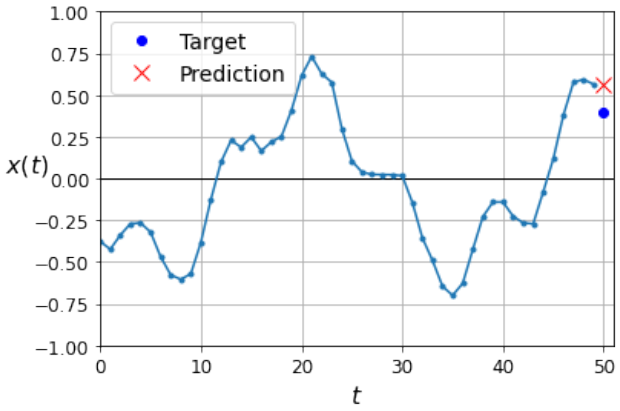

Figure 4.6. Forecasting using previous value.

So, in each one of the hidden layers there are 50x50 weights parameters, plus 50 bias parameters. In the output layer the single neuron is connected to all the 50 neurons from the previous layer, thus there are 50 parameters, plus 1 bias term.

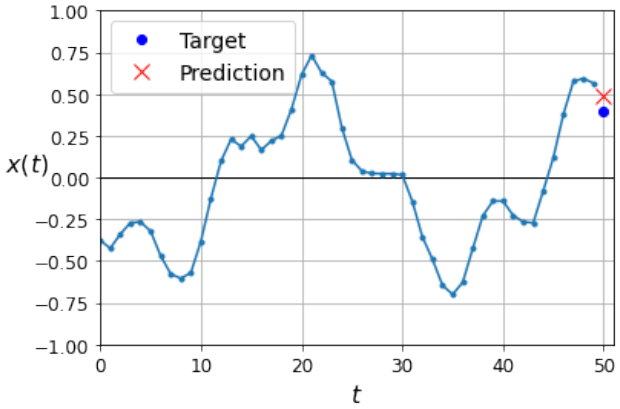

Figure 4.7. Forecast using feedforward neural networks.

The model is trained for 20 epochs, and from Figure 4.7 it is possible to see that the prediction error for this sample looks good and the MSE, equal to 0.003 is way better than the one of the previous approach.

However, a feedforward neural network does not recognize that X is an ordered sequence. It just processes the input set as 50 features and it learns that the prediction is usually close to the 50^{th} element of the series.

The next approach uses RNNs, for which the idea is fundamentally different: there is only 1 feature which is a sequence of length 50.

The simple RNN model is composed by two hidden layers with 20 cells each and the output is again one simple neuron as the prediction is still for one element. The model has now 1281 parameters, much less than the ones of the FFNN.

In case of RNN, the number of parameters is calculated differently:

$$\#parameters = g[h(h+i)+h] \tag{4.12}$$

where g is the number of FFNNs in a unit (simple RNN has 1, GRU has 3, LSTM has 4), h size of hidden units, and i dimension/size of input. So, in this case the first hidden layer has 440 = 1 x [20(20+1) + 20] parameters to estimate, the second hidden layer 820 = 1 x [20(20 + 20) + 20] parameters, and the output layer the remaining 21 = 20 + 1, as the output is only one value.

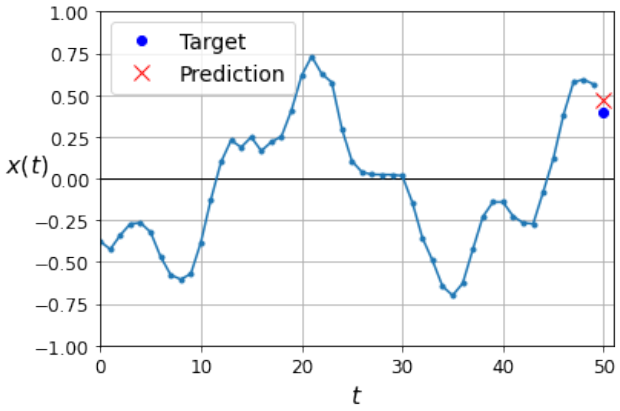

Figure 4.8. Forecast using simple recurrent neural networks.

After 20 epochs the prediction loss is 0.0026, better than the one of the FFNN approach, and the prediction for the same example looks much better too. The thing to notice is that this model had this performance with roughly 80% fewer parameters.

Until now the models only look for one period ahead. Now the memory of RNNs

will be exploited to predict several periods ahead. The first approach uses the RNN model trained before and creates an iterative algorithm to make a prediction for ten periods ahead. The algorithm is summarized as:

- Prediction of the next value
- The prediction is added to the series
- The model is used again to predict a new next value
- Repeat from the second point on until ten elements are added to the series

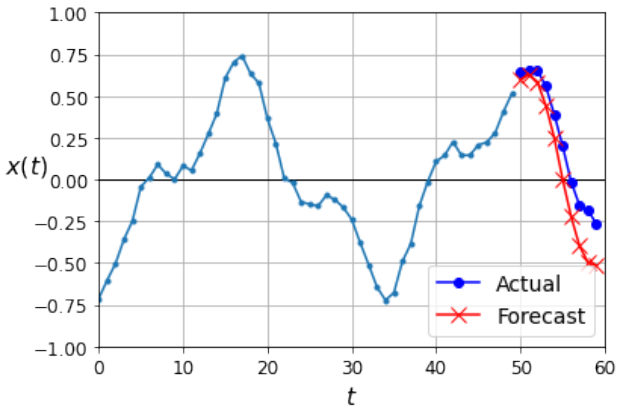

Figure 4.9. Forecasting 10 periods ahead using an iterative algorithm.

The result, shown in Figure 4.9, does not look too bad, but it is easy to notice that the error increases with time, which is reasonable as the model was trained to predict only one period ahead each time.

In order to predict ten more values using RNNs, a new dataset has to be generated. Everything will be the same as before, but this time the output will have ten elements instead of just one, so each time series sample will be composed by 50 elements as input, plus 10 elements as output.
The dataset train-test split remains the same as before.

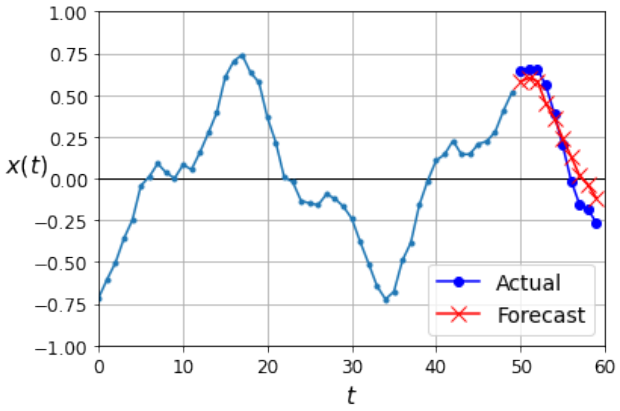

Figure 4.10. Forecasting 10 periods ahead using RNNs.

The deep RNN model has again two hidden layers with 20 units each, but this time the output layer has 10 neurons.

The model has 1470 parameters, and it is still trained for 20 epochs. As shown in Figure 4.10, the quality of the prediction looks better than the single-period model: the error does not diverge with the number of periods.

The next model is a sequence to sequence (seq2seq) model, and it introduces a family of models commonly used for translations and time series prediction.

This model predicts the next 10 periods at each period of the sequence: it forecasts periods 1 to 10 at period 0, and so on until it predicts periods 50 to 59 at the 49^{th} period.

The input data X remains unchanged, but y is now generated as the 10 periods following each and every element of X. To do this, all the 10000 sample time series will have 50 vectors (one for each time period), each with length 10 (prediction horizon).

This is the best model since now. The final period MSE is 0.0085, and for the example considered the prediction looks very good.

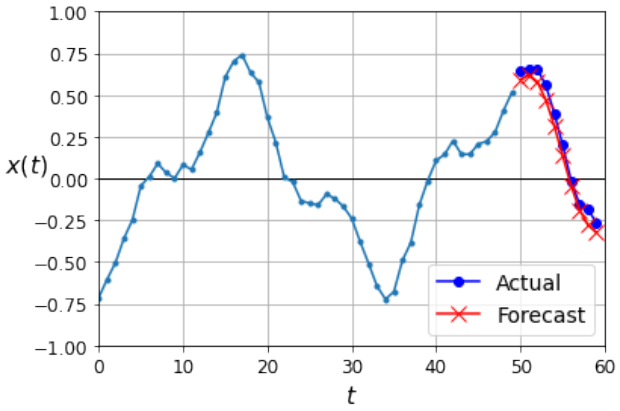

Figure 4.11. Forecasting 10 periods ahead using a seq2seq model.

Lastly, the steps are repeated one more time finally using LSTMs. The deep LSTM shows a very good performance and has almost identical results to the deep sequence to sequence model: the final MSE is 0.0086.

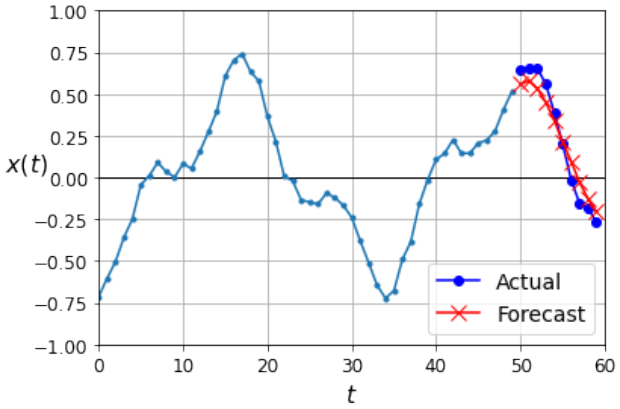

Figure 4.12. Forecasting 10 periods ahead using LSTMs.

Finally, this section has shown the power of LSTMs in comparison with basic neural networks models and simple recurrent neural networks models. In the final chapter of this work LSTMs will be applied to the real-life problem of predicting volatility.

Chapter 5

Application to Financial Time Series - VSTOXX EUR Index

5.1 Data Retrieving

After a good introduction of neural networks, and in particular of recurrent neural networks, it is now time to effectively test the predictive power of these kind of machine learning models. In order to do this, the time series used is the one reporting the VSTOXX EUR Index performance. The VSTOXX Index has been introduced in the first chapter and it is daily monitored by millions of people worldwide. The data has been retrieved directly from the Stoxx website and consists of the time series of the VSTOXX EUR Index daily level from January 4^{th} 1999 to December 30^{th} 2022. A summary of the main descriptive statistics is proposed in the table below:

	Value		Value
count	6110.00	mean	23.93
std	9.45	min	10.68
25%	17.36	50%	22.10
75%	27.55	max	87.51

Figure 5.1 shows the performance of the index in this period.

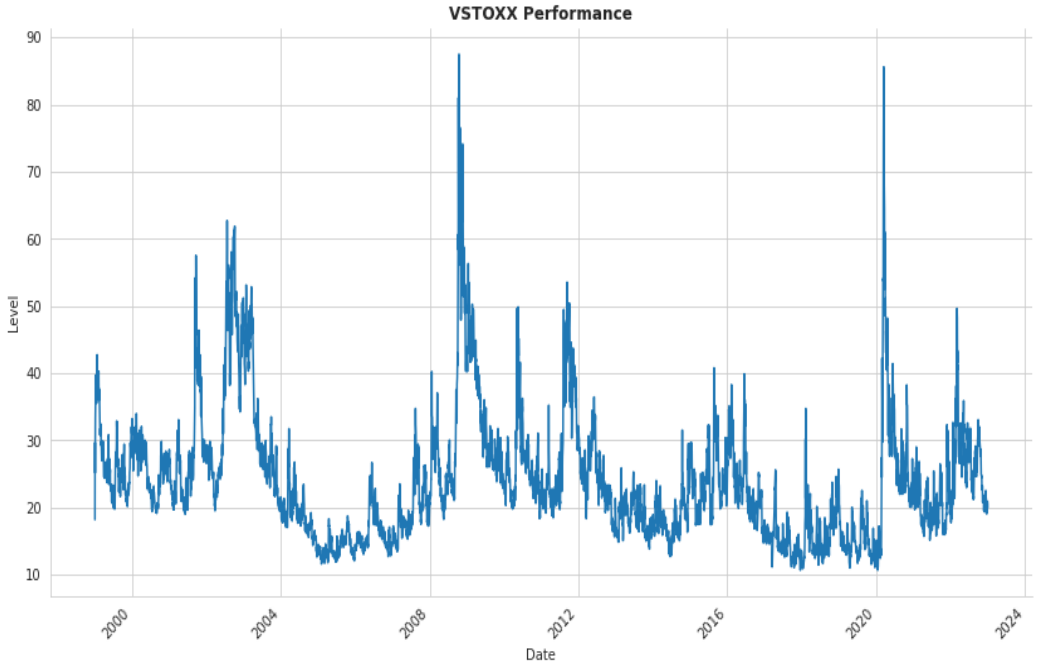

Figure 5.1. Performance of the VSTOXX EUR Index.

5.2 Data Processing

Before talking about data manipulation, it is important to specify that to train the model the dataset is normalized, which means that the data is rescaled from the original range so that all the values are within the range of 0 and 1, included.

It has been introduced before that to feed time series data to recurrent neural networks, a proper shaping is needed.
In this case, the time series is composed by 6110 values:

$$x_1, x_2, ..., x_T \quad where \quad T = 6110 \tag{5.1}$$

The aim is to construct with the dataset available a rolling set of input/output so that the model can learn to predict the next value of the series at each step. The

images below give and idea about the functioning of this sliding window of data:

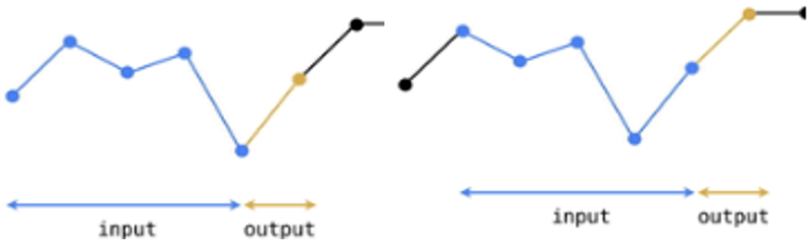

Figure 5.2. Example of rolling time series.

To create this new sliding dataset, 6110 sequences are generated.
Every one of these 6110 sequences has 63 elements, representing approximately three months in trading days. The input/output pairs are shown as the following:

$$\begin{aligned}
&input: \quad x_1, x_2, ..., x_{63} \quad output: \quad x_{64} \\
&input: \quad x_2, x_3, ..., x_{64} \quad output: \quad x_{65} \\
&input: \quad x_3, x_4, ..., x_{65} \quad output: \quad x_{66} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad ... \\
&input: \quad x_{6047}, x_{6048}, ..., x_{6109} \quad output: \quad x_{6110}
\end{aligned}$$

So that for any x_t:

$$x_t = f(x_{t-1}, x_{t-2}, ..., x_{t-S}) \quad \forall t = S+1, ..., T \tag{5.2}$$

The new dataset is then split in train-test. The training data goes from the first observation, the one of January 4^{th} 1999, to the last one of 2019, while the test set goes from the beginning of 2020 to the end of 2022, including three full trading years. Given the very volatile and unpredictable nature of the data during the test period,

the power of this type of neural networks will be highlighted even more.

Recall that the input of any RNN layer must be a matrix of three dimensions: batch_size x n_steps x input_dimension (features). A visual representation is proposed in Figure 5.3:

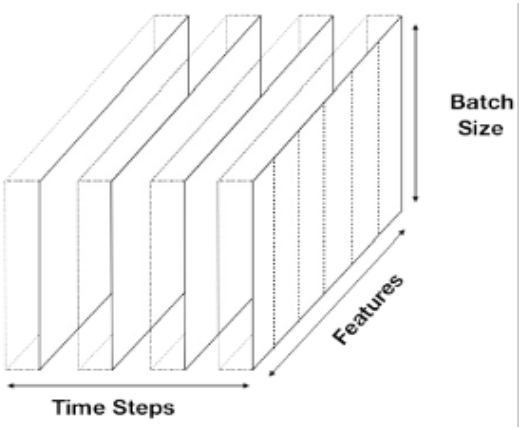

Figure 5.3. Three dimensions of a RNN input.

In this case the batch size, which refers to the number of samples in the train set, is 5279, the time steps are 63, and the dimension, or feature, is only 1, as the closing price is the only feature considered.

5.3 Model Architecture and Performance

The architecture of the model remains similar to the one used in the previous chapter: there are two LSTM layers with 20 units each, and a linear output layer with just one fully connected neuron and a linear activation function. The loss function is again the mean squared error. The model is trained for 100 epochs, but an early stopping is set. Early stopping is used when training neural networks models to avoid overfitting or underfitting. This is because too many epochs can lead to overfit the training set, while few epochs can lead to underfit the training set.

Therefore, by using an early stopping it is possible to specify a relatively large number of epochs, and, at the same time, make sure that the training would stop if

the model performance does not get better in the validation set.

The loss history is presented in figure 5.4. The loss history consists in the 5-epoch rolling average of the training and validation RMSE. It is possible to notice that the training stops at the 54^{th} epoch, but also that the model does not really improve from epoch 20 on.
The best epoch has a loss of 3.1491%, as highlighted by the black dotted vertical line.

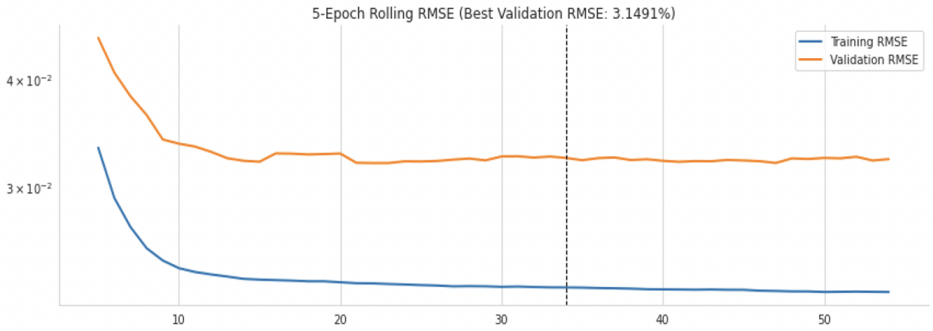

Figure 5.4. Loss history.

5.4 Results

Before showing the results, the data is rescaled to provide an appropriate representation.

The performance of the model can be summarized in the four figures below.
On the upper-left corner there is an outlook of how the model nicely fit the predictions even in the test set.

On the lower-left corner a scatter plot of predictions and actual value of the index is presented. To evaluate this relation the information coefficient (IC) is used. The information coefficient is normally used in the contest of stock return predictions and describes the correlation between predictions and actual values. The coefficient ranges from -1 to 1, included. Therefore, an IC of 95.55% for the test set can be

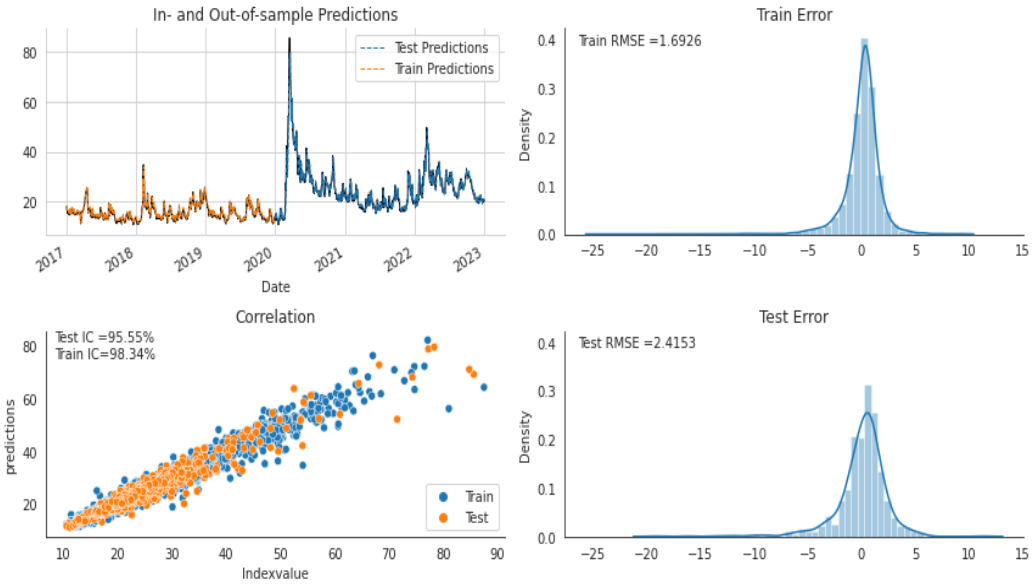

Figure 5.5. LSTM performance on VSTOXX Index predictions.

seen as a satisfying result.

On the right part of the figure there are the distributions for the train and test errors. It is possible to notice that they almost perfectly fit the normal distribution, meaning that the error mean for both sets is very close to 0.

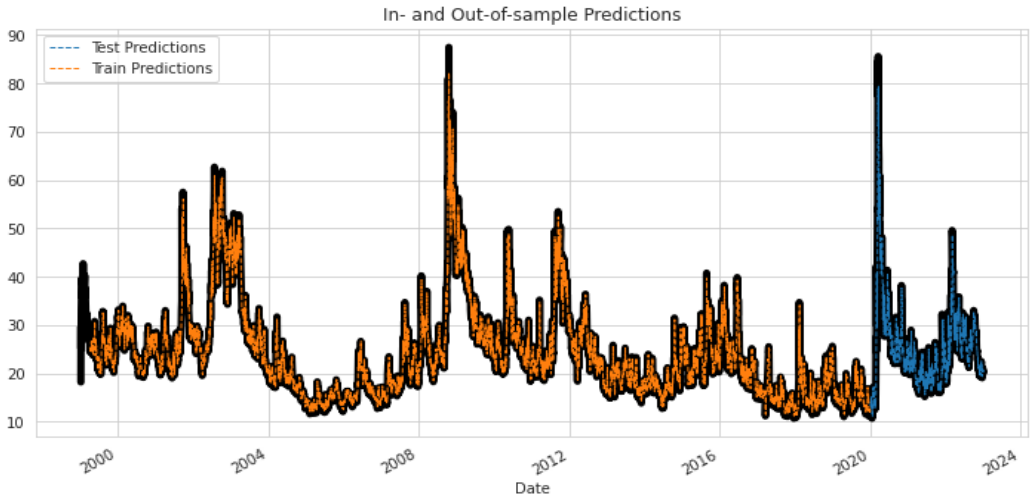

Figure 5.6. In and out-of sample VSTOXX Index predictions.

Figure 5.6 takes a closer look at the predictions in and out-of-sample.

Notice that the larger black line in the background is the real value of the VSTOXX

Index for the period studied. With this said, it is easy to notice that this model highlights the strength of artificial neural networks which use LSTMs. Even if the index seems to be very volatile, the model is able to catch daily movements quite accurately. To provide a last example, the table below reports the predictions for the index value for the last five observations of the test dataset:

Date	Index Value	Test Predictions
2022-12-23	20.04	19.90
2022-12-27	20.66	20.07
2022-12-28	20.51	20.64
2022-12-29	19.93	20.56
2022-12-30	20.89	20.01

Further steps of this work could be to structure, backtest, and implement an algorithmic trading strategy. Once a model has proven to be remarkably reliable, it is possible to put it at work to earn money. For example, given the model created in this chapter, it would be reasonable to create a strategy for which, given the previous three months values of the index, if the value of tomorrow is predicted to be higher than the last value of the time series (today's value), a long position can be open, otherwise a short position can be open. After building a strategy it is possible to backtest it by looking at the returns the strategy would have had for a period in the past. The final step would then be to take into account transaction costs, capital invested, etc.

Chapter 6

Conclusion

In this thesis the aim has been to use a machine learning model to deal with a long-standing problem: predicting volatility.

This work initially focused on the theory behind volatility in finance and the theory behind machine learning. It has been shown how machine learning is replacing the most common approaches to deal with financial problems, especially forecasting.

The outperformance which characterizes machine learning models is explained by the capability of artificial neural networks to catch and exploit non-linear relationships between input and output features, something new compared to previous econometrics models.

The level is increased when using recurrent neural networks. This is because not only these models can detect non-linear relationships, but also can exploit information from past observation. It has been shown how the memory of RNNs is constructed with LSTMs, and how models using these cells outperform standard models.

After the theoretical part, this thesis has focused on the empirical research: forecasting the VSTOXX EUR Index with a machine learning model using LSTMs. The good performance of the model is confirmed by the information coefficients for the train set (98.34%), and for the test set (95.55%), and also by the distributions of the train and test error.

On a final note, there is a citation by Dave Waters, professor at Oxford University, that perfectly fit the purpose of this thesis: "*Predicting the future isn't magic, it's artificial intelligence*".

Bibliography

[1] Abadi, M. (2016), Tensorflow: A System for Large-Scale Machine Learning, "*Proceedings of the* 12^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*"

[2] Barone, E., Barone, G. (2022), Tracking The EURO STOXX 50, "*SSRN: https://ssrn.com/abstract=4203380 or http://dx.doi.org/10.2139/ssrn.4203380*"

[3] Barone, E., Barone, G. (2022), Unscrambling Codes: From Hieroglyphs to Market News, "*International Journal on Natural Language Computing (IJNLC), 11(6), SSRN: https://ssrn.com/abstract=4049797 or http://dx.doi.org/10.2139/ssrn.4049797*"

[4] Abraham, A. (2005), Artificial neural networks, "*Handbook of measuring system design*"

[5] Bhandari, H. N. (2022), Predicting stock market index using LSTM, "*Machine Learning with Applications, 9*"

[6] Bengio, Y., LeCun, Y., Hinton, G. (2015), Deep Learning, "*Nature, 521*"

[7] Castellani, G., De Felice, M., Moriconi, F. (2006), "*Manuale di finanza. Vol.3: Modelli Stocastici e Contratti Derivati*"

[8] Chao, Y., Guo, Z., Xian, L. (2019), Time Series Data Prediction Based on Sequence to Sequence Model, "*IOP Conf. Ser.: Mater. Sci. Eng. 692*"

[9] Dennison, T. (2021), Understanding and Using VSTOXX® Futures

[10] De Spiegeleer, J. (2018), Machine Learning for Quantitative Finance: Fast Derivative Pricing, Hedging and Fitting, "*SSRN: https://ssrn.com/abstract=3191050*"

[11] Engle, R. F., Patton, A. J. (2001), What good is a volatility model?, "*Quantitative Finance, 1*", 237-245

[12] Fischer, T., Krauss, C. (2018), Deep learning with long short-term memory networks for financial market predictions, "*European Journal of Operational Research*"

[13] Goodfellow, I., Bengio, Y., Courville, A. (2016), "*Deep Learning*"

[14] 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 Science, 10(11)*"

[15] Heaton J. B., Polson, N. G., Witte, J. H. (2016), Deep Learning in Finance, "*Applied Stochastic Models in Business and Industry 33(1)*"

[16] Hewamalage, H., Bergmeir, C., Bandara, K. (2020), Recurrent Neural Networks for Time Series Forecasting: Current Status and Future Directions, "*International Journal of Forecasting, Elsevier, 37(1)*", 388-427.

[17] Hull, J. (2018), "*Options, futures, and other derivatives*"

[18] Hull, J. (2020), "*Machine learning in business: An introduction to the world of data science*"

[19] Jansen, S. (2020), "*Machine Learning for Algorithmic Trading*", 2^{nd} Edition

[20] Kandel, I., Castelli, M. (2020), Transfer Learning with Convolutional Neural Networks for Diabetic Retinopathy Image Classification. A Review, "*Applied Science, 10(6)*"

[21] Lipton, Z. C. (2015), A Critical Review of Recurrent Neural Networks for Sequence Learning

[22] Loiseau, J. (2019). Rosenblatt's perceptron, the first modern neural network, "*https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a*"

[23] McCulloch, W. S., Pitts, W. (1943), A logical calculus of the ideas immanent in nervous activity, "*The bulletin of mathematical biophysics, 5(4)*", 115-133

[24] McMillan, L.G. (2002), "*Options as a Strategic Investment*", 4^{th} Edition

[25] Mixon, S. (2011), What Does Implied Volatility Skew Measure?, "*The Journal of Derivatives, 18(4)*"

[26] Qontigo (2023), STOXX STRATEGY INDEX GUIDE

[27] Oliver Muncharaz, J. (2020), Comparing classic time series models and the 137 LSTM recurrent neural network: An application to SP 500 stocks, "*Finance, Markets and Valuation 6(2)*", 137-148

[28] Raudys, A., Goldstein, E. (2022), Forecasting Detrended Volatility Risk and Financial Price Series Using LSTM Neural Networks and XGBoost Regressor, "*Journal of Risk and Financial Management 15: 602*"

[29] Rosenblatt, F. (1957), The Percepton, a Perciving and Recognizing Automaton.

[30] Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986). Learning representations by back-propagating errors, "*Nature, 323(6088)*", 533–536.

[31] Werbos, P. J. (1990), Backpropagation Through Time: What It Does and How to Do It, "*Proceedings of the IEEE, 78(10)*", 1550-1560

Appendix - Python Code

6.1 Predicting Noisy Sine Wave

6.1.1 Import Packages

```
import sklearn
import tensorflow as tf

# Common imports
import pandas as pd
import numpy as np
import os

from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler

# to make this notebook's output stable across runs
np.random.seed(42)
tf.random.set_seed(42)


# To plot figures
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)
```

```
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline

from pathlib import Path

import numpy as np
import pandas as pd
import pandas_datareader.data as web
from scipy.stats import spearmanr

from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow import keras

import matplotlib.pyplot as plt
import seaborn as sns
```

6.1.2 Helper Functions

```
def plot_series(series, y=None, y_pred=None, x_label="$t$", y_label="
                                    $x(t)$", legend=True):
plt.plot(series, ".-")
if y is not None:
    plt.plot(n_steps, y, "bo", label="Target")
if y_pred is not None:
    plt.plot(n_steps, y_pred, "rx", markersize=10, label="Prediction"
                                        )
plt.grid(True)
if x_label:
    plt.xlabel(x_label, fontsize=16)
if y_label:
    plt.ylabel(y_label, fontsize=16, rotation=0)
plt.hlines(0, 0, 100, linewidth=1)
plt.axis([0, n_steps + 1, -1, 1])
if legend and (y or y_pred):
    plt.legend(fontsize=14, loc="upper left")
```

```
def plot_learning_curves(loss, val_loss):
plt.plot(np.arange(len(loss)) + 0.5, loss, "b.-", label="Training
                                    loss")
plt.plot(np.arange(len(val_loss)) + 1, val_loss, "r.-", label="
                                    Validation loss")
plt.gca().xaxis.set_major_locator(mpl.ticker.MaxNLocator(integer=True
                                    ))
plt.axis([1, 20, 0, 0.05])
plt.legend(fontsize=14)
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.grid(True)
```

```
def plot_multiple_forecasts(X, Y, Y_pred):
n_steps = X.shape[1]
ahead = Y.shape[1]
plot_series(X[0, :, 0])
plt.plot(np.arange(n_steps, n_steps + ahead), Y[0, :, 0], "bo-",
                                    label="Actual")
plt.plot(np.arange(n_steps, n_steps + ahead), Y_pred[0, :, 0], "rx-",
                                    label="Forecast", markersize=10)
plt.axis([0, n_steps + ahead, -1, 1])
plt.legend(fontsize=14)
```

6.1.3 Generate the Dataset

```
np.random.seed(42) #to receive always the same outputs
def generate_time_series(batch_size, n_steps):
freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
time = np.linspace(0, 1, n_steps)
series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #
                                    wave 1
series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # +
                                    wave 2
series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # +
                                    noise
return series[..., np.newaxis].astype(np.float32)
```

```
batch_size = 10000
n_steps = 50

series = generate_time_series(batch_size, n_steps+1)

# the shape of "series" is (10000,51,1)

X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]

#plot any of the series
col = 1822 #chooses the 1822th series from the 7000 created
```

```
plot_series(X_train[col, :, 0], y_train[col, 0])
plt.show()
```

6.1.4 Naive Prediction Using Last Observed Value

```
from tensorflow import keras
y_pred = X_valid[:, -1] #the prediction is the last value of the ith
                                    time series in X_valid (so the
                                    50th time step)
np.mean(keras.losses.mean_squared_error(y_valid, y_pred))

i = 0 #the first sample is taken

#show the sample that will be used for all the other approaches
plot_series(X_valid[i, :, 0], y_valid[i, 0], y_pred[i, 0])
plt.show()
```

6.1.5 Prediction Using FFNN

```
#create the model
np.random.seed(42)
tf.random.set_seed(42)
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(50),
    keras.layers.Dense(50),
    keras.layers.Dense(1)
])
model.summary()
model.compile(loss="mse", optimizer="adam")
```

```
#train the model
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

```
#evaluate the model
model.evaluate(X_valid, y_valid)

plot_learning_curves(history.history["loss"], history.history["
                                    val_loss"])
plt.show()

y_pred = model.predict(X_valid)
plot_series(X_valid[0, :, 0], y_valid[0, 0], y_pred[0, 0])
plt.show()
```

6.1.6 Prediction Using RNN

```
#create the model
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)])
model.summary()
```

```
#train the model
model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, y_train, epochs=20,
                    validation_data=(X_valid, y_valid))
```

```
#evaluate the model
model.evaluate(X_valid, y_valid)

plot_learning_curves(history.history["loss"], history.history["
                                    val_loss"])
plt.show()

y_pred = model.predict(X_valid)
plot_series(X_valid[0, :, 0], y_valid[0, 0], y_pred[0, 0])
plt.show()
```

6.1.7 One-Period Model Forecasting Several Steps Ahead

```
np.random.seed(43) #the sample is changed

#create the new dataset
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new

#create the model
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)
Y_pred = X[:, n_steps:]
#at every step the model predicts the next step (first one would
                                    predict the 51st term then 52nd
                                    until 60th term) using the ones
                                    done before
```

```
#show the performance
plot_multiple_forecasts(X_new, Y_new, Y_pred)
plt.show()
```

6.1.8 RNN Model Forecasting Several Steps Ahead

```
#create new dataset
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]   #
                                    X_train is from 0 to 50 while
                                    Y_train is from 51 to 60
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10
                                    :, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

```
#create model
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
model.summary()
```

```
#train the model
model.compile(loss="mse", optimizer="adam")
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

```
#show the performance
np.random.seed(43)

series = generate_time_series(1, 50 + 10)
X_new, Y_new = series[:, :50, :], series[:, -10:, :]
Y_pred = model.predict(X_new)[..., np.newaxis] #WE USED THE MODEL WE
                                    JUST TRAINED TO MAKE A PREDICTION

plot_multiple_forecasts(X_new, Y_new, Y_pred)
plt.show()
```

6.1.9 Sequence to Sequence Model Forecasting Several Steps Ahead

```
#generate new dataset
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train = series[:7000, :n_steps]
X_valid = series[7000:9000, :n_steps]
X_test = series[9000:, :n_steps]

Y = np.empty((10000, n_steps, 10))
for step_ahead in range(1, 10 + 1):
    Y[..., step_ahead - 1] = series[..., step_ahead:step_ahead +
                                        n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:,
                                            -1])
```

```
#create the model
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[
                                        None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.summary()
```

```
#train the model
model.compile(loss="mse", optimizer=keras.optimizers.Adam(
                                    learning_rate=0.01),
              metrics=[last_time_step_mse])

history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

```
#evaluate the model
model.evaluate(X_valid, Y_valid)
np.random.seed(43)

series = generate_time_series(1, 50 + 10)
X_new, Y_new = series[:, :50, :], series[:, 50:, :]
Y_pred = model.predict(X_new)[:, -1][..., np.newaxis]

plot_multiple_forecasts(X_new, Y_new, Y_pred)
plt.show()
```

6.1.10 LSTM Model Forecasting Several Steps Ahead

```
#create the model
np.random.seed(42)
tf.random.set_seed(42)

model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1
                                        ]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.summary()
```

```
#train the model
model.compile(loss="mse", optimizer="adam", metrics=[
                                    last_time_step_mse])
history = model.fit(X_train, Y_train, epochs=20,
                    validation_data=(X_valid, Y_valid))
```

```
#evaluate the model
model.evaluate(X_valid, Y_valid)

plot_learning_curves(history.history["loss"], history.history["
                                    val_loss"])
plt.show()

np.random.seed(43)

series = generate_time_series(1, 50 + 10)
X_new, Y_new = series[:, :50, :], series[:, 50:, :]
Y_pred = model.predict(X_new)[:, -1][..., np.newaxis]

plot_multiple_forecasts(X_new, Y_new, Y_pred)
plt.show()
```

6.2 Forecasting the VSTOXX EUR Index with a RNN

6.2.1 Import Packages

```
import warnings
warnings.filterwarnings('ignore')
%matplotlib inline

from pathlib import Path

import numpy as np
import pandas as pd
import pandas_datareader.data as web
from scipy.stats import spearmanr

from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import MinMaxScaler

import tensorflow as tf
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM
from tensorflow import keras

import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style('whitegrid')
np.random.seed(42)
```

6.2.2 Data Retrieve

```
V2X = pd.read_excel("/content/sample_data/h_v2tx_DEC22.xls")

V2X = V2X.set_index('Date')

ax = V2X.plot(title='V2X',
              legend=False,
              figsize=(14, 8),
              rot=45)
ax.set_xlabel('Date')
ax.set_ylabel('Level')
plt.title("VSTOXX Performance", fontweight='bold')
sns.despine()
```

6.2.3 Data Processing

```
scaler = MinMaxScaler()

V2X_scaled = pd.Series(scaler.fit_transform(V2X).squeeze(), index=V2X
                                    .index)
V2X_scaled.describe()
```

```
def create_univariate_rnn_data(data, window_size):
    n = len(data)
    y = data[window_size:]
    data = data.values.reshape(-1, 1) # make 2D
    X = np.hstack(tuple([data[i: n-j, :] for i, j in enumerate(range(
                                        window_size, 0, -1))]))
    return pd.DataFrame(X, index=y.index), y
# We apply this function to the rescaled stock index for a
                                    window_size=63 to obtain a two-
                                    dimensional dataset of shape
                                    number of samples x number of
                                    timesteps:

window_size = 63

X, y = create_univariate_rnn_data(V2X_scaled, window_size=window_size
                                    )
```

```
#Train-test split
X_train = X[:'2019'].values.reshape(-1, window_size, 1)
y_train = y[:'2019']

# keep three years for testing
X_test = X['2020':'2022'].values.reshape(-1, window_size, 1)
y_test = y['2020':'2022']

#set the correct shape for the data
n_obs, window_size, n_features = X_train.shape
```

6.2.4 Create LSTM Model

```
LSTM = Sequential([
    LSTM(units=20, return_sequences = True, input_shape=(window_size,
                                        n_features), name='LSTM'),
    LSTM(units=20, input_shape=(window_size, n_features), name='LSTM1
                                        '),
    Dense(1, name='Output')
])

LSTM.summary()
```

6.2.5 Train the model

```
optimizer = keras.optimizers.RMSprop(lr=0.001, rho=0.9, epsilon=1e-08
                                    , decay=0.0)

LSTM.compile(loss='mean_squared_error', optimizer=optimizer)

early_stopping = EarlyStopping(monitor='val_loss', patience=20,
                                    restore_best_weights=True)

lstm_training = LSTM.fit(X_train,
                         y_train,
                         epochs=100,
                         batch_size=20,
                         shuffle=True,
                         validation_data=(X_test, y_test),
                         callbacks=[early_stopping],
                         verbose=1)
```

6.2.6 Performance of the model

```
fig, ax = plt.subplots(figsize=(12, 4))

loss_history = pd.DataFrame(lstm_training.history).pow(.5)
loss_history.index += 1
best_rmse = loss_history.val_loss.min()

best_epoch = loss_history.val_loss.idxmin()

title = f'5-Epoch Rolling RMSE (Best Validation RMSE: {best_rmse:.4%}
                                    )'
loss_history.columns=['Training RMSE', 'Validation RMSE']
loss_history.rolling(5).mean().plot(logy=True, lw=2, title=title, ax=
                                    ax)

ax.axvline(best_epoch, ls='--', lw=1, c='k')

sns.despine()
fig.tight_layout()
```

```
train_rmse_scaled = np.sqrt(LSTM.evaluate(X_train, y_train, verbose=0
                                    ))
test_rmse_scaled = np.sqrt(LSTM.evaluate(X_test, y_test, verbose=0))
print(f'Train RMSE: {train_rmse_scaled:.4f} | Test RMSE: {
                                    test_rmse_scaled:.4f}')
```

```
train_predict_scaled = LSTM.predict(X_train)
test_predict_scaled = LSTM.predict(X_test)
```

```
train_ic = spearmanr(y_train, train_predict_scaled)[0]
test_ic = spearmanr(y_test, test_predict_scaled)[0]
print(f'Train IC: {train_ic:.4f} | Test IC: {test_ic:.4f}')
```

6.2.7 Predictions Rescaling

```
train_predict = pd.Series(scaler.inverse_transform(
                                    train_predict_scaled).squeeze(),
                                    index=y_train.index)
test_predict = (pd.Series(scaler.inverse_transform(
                                    test_predict_scaled).squeeze(),
                                    index=y_test.index))

y_train_rescaled = scaler.inverse_transform(y_train.to_frame()).
                                    squeeze()
y_test_rescaled = scaler.inverse_transform(y_test.to_frame()).squeeze
                                    ()
```

```
train_rmse = np.sqrt(mean_squared_error(train_predict,
                                    y_train_rescaled))
test_rmse = np.sqrt(mean_squared_error(test_predict, y_test_rescaled)
                                    )
f'Train RMSE: {train_rmse:.2f} | Test RMSE: {test_rmse:.2f}'
```

```
V2X['Train Predictions'] = train_predict
V2X['Test Predictions'] = test_predict
V2X = V2X.join(train_predict.to_frame('predictions').assign(data='
                                    Train').append(test_predict.
                                    to_frame('predictions').assign(
                                    data='Test')))
```

6.2.8 Plot Results

```
fig=plt.figure(figsize=(14,7))
ax1 = plt.subplot(221)

V2X.loc['2017':, 'Indexvalue'].plot(lw=1, ax=ax1, c='k')
V2X.loc['2017':, ['Test Predictions', 'Train Predictions']].plot(lw=1
                                    , ax=ax1, ls='--')
ax1.set_title('In- and Out-of-sample Predictions')


with sns.axes_style("white"):
    ax3 = plt.subplot(223)
    sns.scatterplot(x='Indexvalue', y='predictions', data=V2X, hue='
                                        data', ax=ax3)
    ax3.text(x=.02, y=.95, s=f'Test IC ={test_ic:.2%}', transform=ax3
                                        .transAxes)
    ax3.text(x=.02, y=.87, s=f'Train IC={train_ic:.2%}', transform=
                                        ax3.transAxes)
    ax3.set_title('Correlation')
    ax3.legend(loc='lower right')

    ax2 = plt.subplot(222)
    ax4 = plt.subplot(224, sharex = ax2, sharey=ax2)
    sns.distplot(train_predict.squeeze()- y_train_rescaled, ax=ax2)
    ax2.set_title('Train Error')
    ax2.text(x=.03, y=.92, s=f'Train RMSE ={train_rmse:.4f}',
                                        transform=ax2.transAxes)
    sns.distplot(test_predict.squeeze()-y_test_rescaled, ax=ax4)
    ax4.set_title('Test Error')
    ax4.text(x=.03, y=.92, s=f'Test RMSE ={test_rmse:.4f}', transform
                                        =ax4.transAxes)

sns.despine()
fig.tight_layout()
```

```
fig, ax = plt.subplots(figsize=(25, 12))

V2X.loc['1999':, 'Indexvalue'].plot(lw=5, ax=ax, c='k')
V2X.loc['1999':, ['Test Predictions', 'Train Predictions']].plot(lw=1
                                    , ax=ax, ls='--')
ax.set_title('In- and Out-of-sample Predictions')
```

Summary

Introduction

In the most recent decades, machine learning and algorithms have played an increasingly important role in influencing the financial sector.

Renaissance Technologies, established in 1982 by the mathematician James Simons, was the first hedge fund to demonstrate its dependence on systematic methods based on algorithms. With the secretive Medallion Fund, which has generated an estimated annualized return of over 35% since 1982, he launched this revolution in investing. Citadel, Two Sigma, D.E. Shaw, and AQR (Applied Quantitative Research) are some of the most well-known and effective quantitative hedge funds basing their strategies on algorithms.

In the course of the thesis, after a deep dive in volatility, machine learning, and neural networks, the focus will be in the creation of a predicting model using recurrent neural networks, the type of neural networks implemented when dealing with sequential data.

Volatility

The historical volatility of an asset is a gauge of its return unpredictability. It is derived using the standard deviation of the logarithmic returns over a specific time period and is expressed as a percentage. The formula is:

$$\sigma = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(R_t - \bar{R})^2} \tag{6.1}$$

Where:

n = number of observations

$R_t = \ln(\frac{S_t}{S_{t-1}})$

$\bar{R}$ = average of R_t for t = 1,..,n

S_t = price at time t

However, the previous formula would refer to the population variance, which would assume to have all the data point of the entire population. As a matter of fact, when measuring the historical volatility, it is used the square root of the sample variance, expressed in Formula 6.2:

$$\sigma_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}(R_t - \bar{R})^2 \tag{6.2}$$

What has been said so far is based on earlier observations. In contrast to historical volatility, implied volatility is a measure that looks forward and serves as a gauge for how volatile an underlying is thought to be going to be in the future. Implied volatility is a measure generated from option pricing, and it is expressed annually and in percentage form.

The structure of an option premium is the sum of the intrinsic value and the extrinsic value. While the extrinsic value is determined by a number of variables, including maturity, dividends (if any), interest rate, and most significantly implied volatility, the intrinsic value just depends on the strike price and the underlying price. So, the higher the implied volatility, the higher the option premium.

There is no set of formulas for calculating implied volatility; rather, it is calculated so that the theoretical price of an option is equal to its market price, which is observable in the market. To determine the theoretical price of European options for stocks that don't pay dividends it is commonly used the Black-Scholes-Merton model, introduced in 1973.

Implied volatility is extremely important in financial markets. Traders monitor implied volatility with two main indexes: the CBOE Volatility Index (VIX), and the VSTOXX EUR Index (V2TX), respectively used for American stocks in the S&P500 Index and European stocks in the EURO STOXX 50 Index. The model which will be built will be applied to the V2TX.

Figure 6.1 illustrates one of the reason why it is important to monitor the VSTOXX

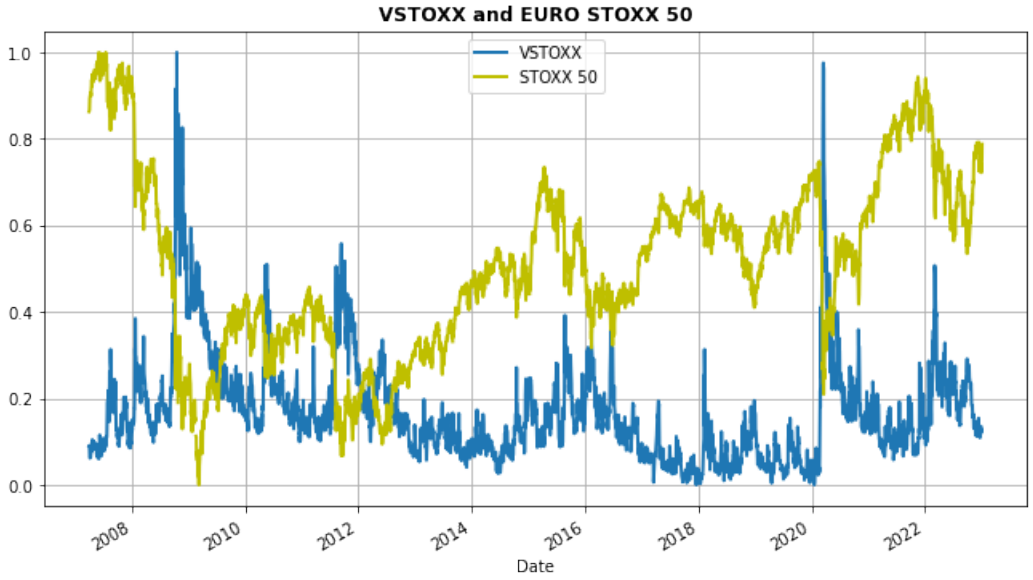

Figure 6.1. VSTOXX Index Performance versus EURO STOXX 50 Index from 2007 to 2022.

Index: whenever the EURO STOXX 50 Index reaches a relative maximum, the V2TX approaches a relative minimum, and vice versa.

Recurrent Neural Networks

Recurrent neural networks (RNNs) are a type of neural networks used to process sequential data where patterns evolve over time and where order matters.

In feedforward neural networks each sample is independent and identically distributed. However, there are some tasks for which the order of data matters. Among those there are speech recognition, translation, image captioning, and time series prediction, which will be the main focus later in this work.

The aim of recurrent neural network is to capture order-dependent information, and to do this, the algorithm requires a sort of memory for preceding data points. The most important innovation with RNNs, which had been introduced in 1986, is that the output is a function of both the previous output and the new input, and thus

the information is incorporated from prior observations. The assumption of RNNs is that the input data gets generated as a sequence and that previous observations are relevant for predicting following values.

In a recurrent neuron, the output is calculated by multiplying the inputs to the weights (and adding a bias). The same output then goes back in the network along with the next input to generate the next output. So, any time after time 0 there are two inputs in a RNN cell: the new input x_t and h_{t-1}. The last term is referred as the hidden state and it is the term for which information from previous observations is preserved in the structure.

A single RNN cell can be described by the following equations:

$$h_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}) \tag{6.3}$$

$$\mathbf{y}_t = g(\mathbf{W}_{yh}\mathbf{h}_t) \tag{6.4}$$

Where W_{xh} is the matrix of weights connecting input with the hidden unit, W_{hh} is the matrix of weights connecting the previous hidden unit with the next hidden unit, and W_{yh} is the matrix of weights connecting the last hidden unit with the output. Usually, the activation functions used are tanh and ReLU. Figure 6.2 represents the functioning of a RNN by showing the unrolled computational graph of a single RNN cell:

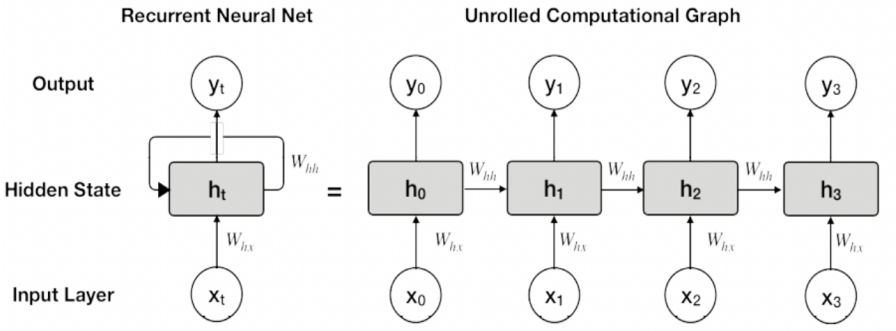

Figure 6.2. Computational graph of a single RNN cell.

Regarding training of recurrent neural networks models, a variation of backprop-

agation is used. The only difference, or complication, arising is that not only the error is propagated through layers, but also through time. As a matter of fact, the process is referred as backpropagation through time (BPTT).

Although the training process does not have computational issues, it may cause the vanishing/exploding gradient problem when sequences in input get longer, something that may prevent the model to catch temporal dependencies. To deal with this problem many approaches have been proposed. However, the most successful use gated units. With these units during the training process the model learns how much past information to retain in the current state and how much information to reset. The most common examples are long-short term memory (LSTM) units, introduced in 1997 by Hochreiter and Schmidhuber, and gated recurrent units (GRU).
LSTMs have different gates which determines whether to keep an information and, if so, how much of that information to keep in the system.
From "outside" the LSTM cell looks the same as a RNN cell, however, the LSTM cell is far more complicated internally, as shown by Figure 6.3:

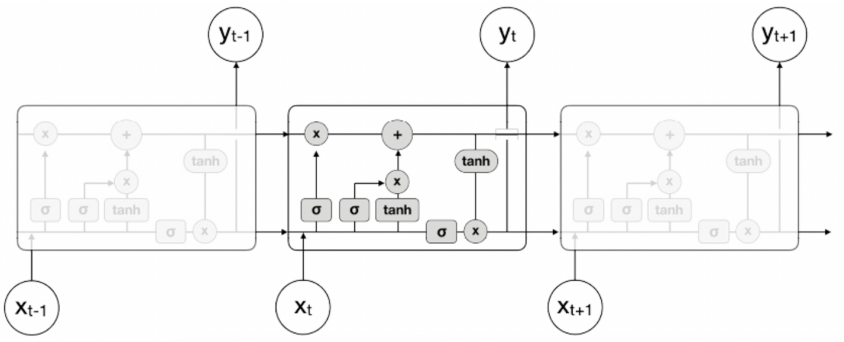

Figure 6.3. Inside the LSTM unit.

Figure 6.4 helps to describe the functioning of the cell.
In Figure 6.4, the dark grey elements represent layers in which weights and biases are learned during training while the white circles symbolize element-wise operations.
The cell state, c, is the one on top of the cell and it has connections with all the

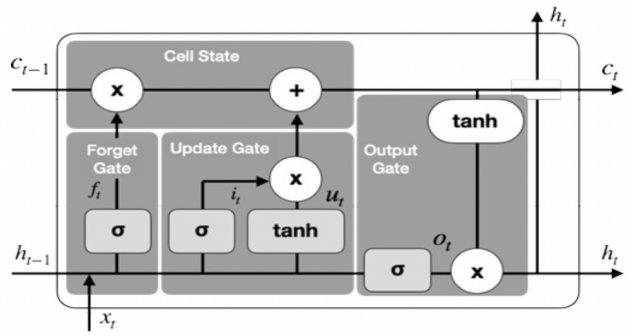

Figure 6.4. Functioning of a LSTM unit.

other gates.

First, the forget gate controls how much information should get in the cell state. It receives two inputs: the hidden state h_{t-1}, and the current input x_t; it combines them and applies a sigmoid activation function:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{6.5}$$

The result will be in a range between 0 and 1, included, and will be multiplied by the cell state c_{t-1}, updating it accordingly.

Then, the update gate receives h_{t-1} and x_t as well and computes a sigmoid and a tanh on them. The two results, u_t and i_t are multiplied and then added or subtracted, depending on the sign, from the cell state.

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{6.6}$$

$$u_t = tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{6.7}$$

Finally, in the output gate h_{t-1} and x_t go through a sigmoid function to output o_t.

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{6.8}$$

At the same time o_t is combined with the ultimate cell state c_t, which is normalized by the tanh function.

The final outputs of the unit are then h_t, which can be thought as the short-term

memory, and c_t, which can be thought as the long-term memory.

$$c_t = f_t \odot c_{t-1} + i_t \odot u_t \tag{6.9}$$

$$h_t = o_t \odot tanh(c_t) \tag{6.10}$$

Predicting the VSTOXX EUR Index by a RNN

After an introduction of recurrent neural networks, it is now time to effectively test the predictive power of this kind of machine learning models. In order to do this, the time series used is the one reporting the VSTOXX EUR Index performance. The data has been retrieved directly from the Stoxx website and consists of the time series of the VSTOXX EUR Index daily level from January 4^{th} 1999 to December 30^{th} 2022. Figure 6.5 shows the performance of the index in this period.

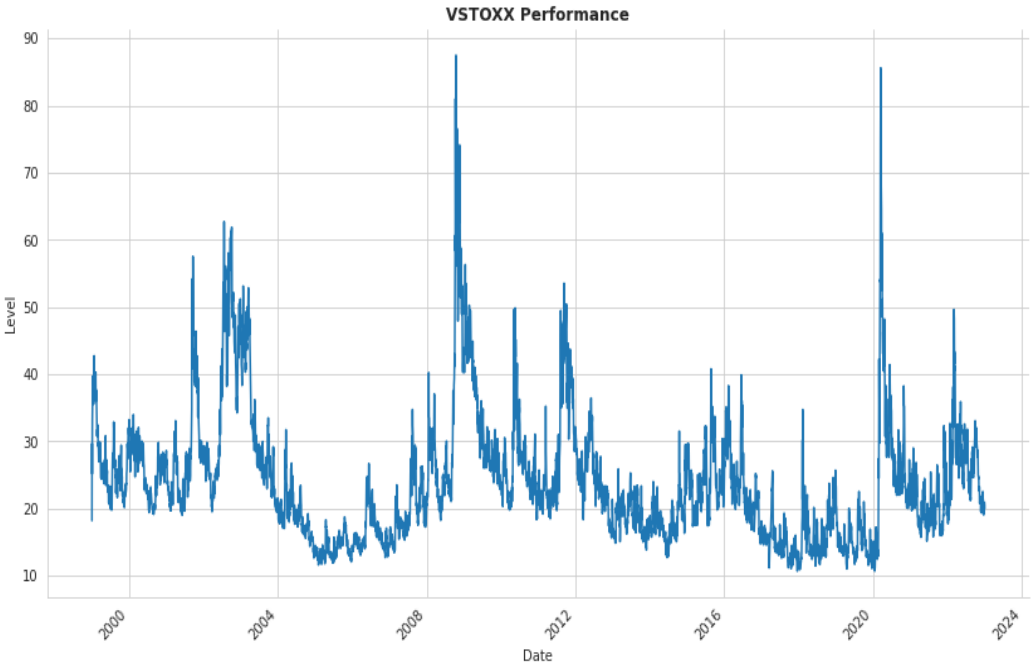

Figure 6.5. Performance of the VSTOXX EUR Index.

A summary of the main descriptive statistics is in the table below:

	Value		Value
count	6110.00	mean	23.93
std	9.45	min	10.68
25%	17.36	50%	22.10
75%	27.55	max	87.51

The dataset is normalized, which means that the data is rescaled from the original range so that all the values are within the range of 0 and 1, included.
In order to feed a recurrent neural network, the dataset has to be shaped in order to make it three-dimensional. So, the dataset is shaped to get a rolling set of input/output so that the model can learn to predict the next value of the series at each step.
The 6110 sequences of the dataset have 63 elements, representing approximately three months in trading days. The input/output pairs are shown as the following:

$$
\begin{aligned}
input: \quad & x_1, x_2, ..., x_{63} \quad output: \quad x_{64} \\
input: \quad & x_2, x_3, ..., x_{64} \quad output: \quad x_{65} \\
input: \quad & x_3, x_4, ..., x_{65} \quad output: \quad x_{66} \\
& \qquad\qquad\qquad\qquad\qquad\quad ... \\
input: \quad & x_{6047}, x_{6048}, ..., x_{6109} \quad output: \quad x_{6110}
\end{aligned}
$$

The new dataset is then split in train-test. The training data goes from the first observation, the one of January 4^{th} 1999, to the last one of 2019, while the test set goes from the beginning of 2020 to the end of 2022, including three full trading years.
The architecture of the model is composed by two LSTM layers with 20 units each, and a linear output layer with just one fully connected neuron and a linear activation function. The loss function is the mean squared error. The model is trained for 100 epochs, but an early stopping is set. Early stopping is used when training neural

networks models to avoid overfitting or underfitting. This is because too many epochs can lead to overfit the training set, while few epochs can lead to underfit the training set.

Therefore, by using an early stopping it is possible to specify a relatively large number of epochs, and, at the same time, make sure that the training would stop if the model performance does not get better in the validation set.

The loss history is presented in Figure 6.6. The loss history consists in the 5-epoch rolling average of the training and validation RMSE. It is possible to notice that the training stops at the 54^{th} epoch, but also that the model does not really improve from epoch 20 on.

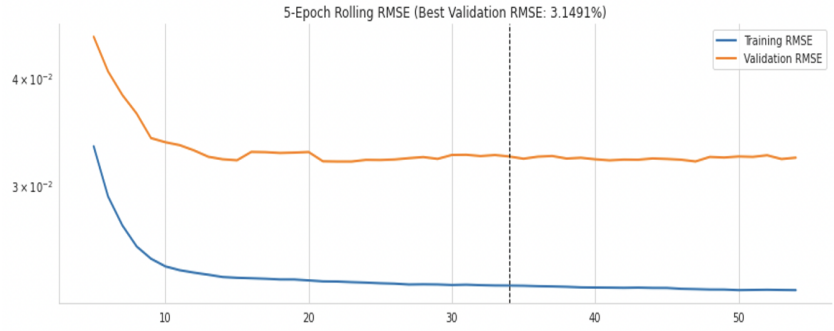

Figure 6.6. Loss history.

Before showing the results, the data is rescaled to provide an appropriate representation.

The performance of the model can be summarized in the four figures below.

On the upper-left corner there is an outlook of how the model nicely fit the predictions even in the test set.

On the lower-left corner a scatter plot of predictions and actual value of the index is presented. To evaluate this relation the information coefficient (IC) is used. The information coefficient is normally used in the contest of stock return predictions and describes the correlation between predictions and actual values. The coefficient ranges from -1 to 1, included. Therefore, an IC of 95.55% for the test set can be

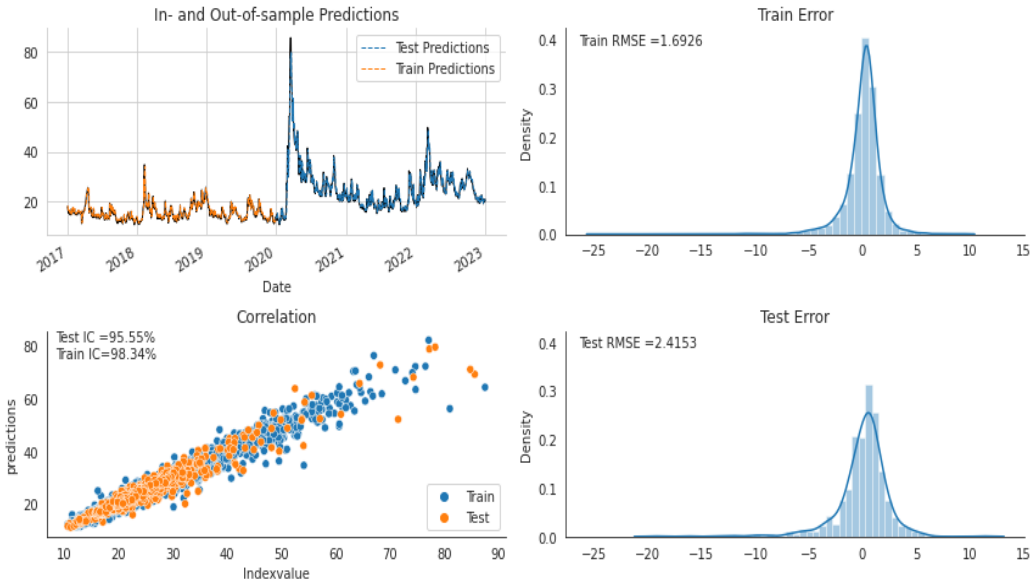

Figure 6.7. LSTM performance on VSTOXX Index predictions.

seen as a satisfying result.

On the right part of the figure there are the distributions for the train and test errors. It is possible to notice that they almost perfectly fit the normal distribution, meaning that the error mean for both sets is very close to 0.

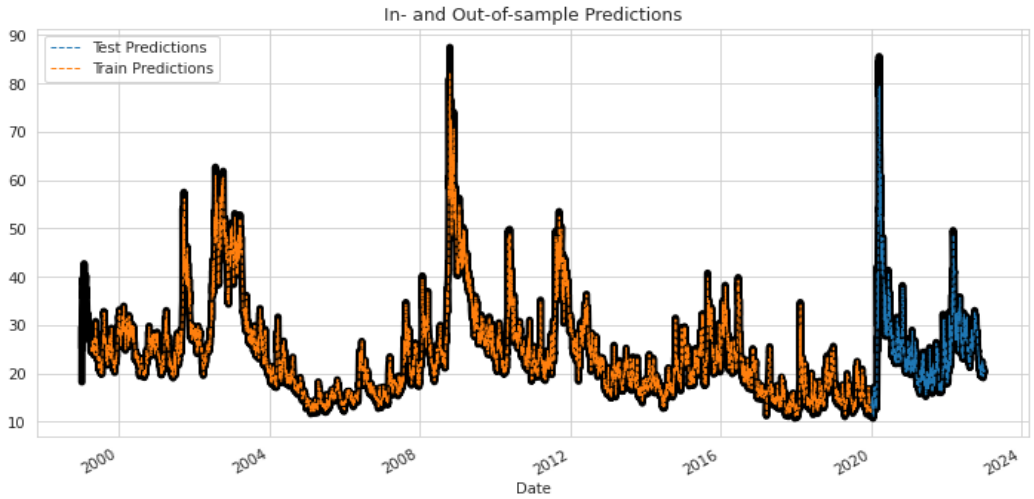

Figure 6.8. In and out-of sample VSTOXX Index predictions.

Figure 6.8 takes a closer look at the predictions in and out-of-sample.

Notice that the larger black line in the background is the real value of the VSTOXX

Index for the period studied. With this said, it is easy to notice that this model highlights the strength of artificial neural networks which use LSTMs.