Università
Ca'Foscari
Venezia

Master's Degree

in Economics and Finance

Final Thesis

Forecasting Stock Index Volatility: a comparison between GARCH and LSTM models

Supervisor
Ch. Prof. Marco Corazza

Graduand
Greta Vitali
868008

Academic Year
2018 / 2019

Acknowledgement

Firstly, I would like to thank my thesis advisor Prof. Marco Corazza for his fundamental guidance and his constant support to my research activity and the development of this thesis.

I also have great pleasure in acknowledging my gratitude to my colleagues from my internship at UBI Banca for their encouragement and for allowing me to collect the data for this research.

I also would like to thank my parents and my brother for giving me the right motivation, for believing in me and for allowing me to continue my studies in Venice.

A sincere thank goes also to my grandparents, my aunt Dede and my uncle Don Gianni for supporting me during this experience.

I would like to thank all my friends and my roommates for listening, offering me advice and sharing with me joys, sacrifices and successes.

A final thank is for Matteo for always being by my side and for the continuous support and encouragement throughout the past five years.

Abstract

The financial world is characterized by the uncertainty of events and this phenomenon can expose operators to huge financial risks. Thus, there is a need to measure this uncertainty, with the aim to predict it and to make adequate plans of action. The concept of uncertainty is often associated with the definition of volatility, which is a measure of the variation of stock prices of a financial instrument during the time. But modelling volatility is not a trivial task, because of the essence of financial stock prices, which usually present volatility clusters, fat tails, nonnormality and structural breaks in the distribution. A popular class of models able to capture many of these stylized facts is the ARCH/GARCH family. As a matter of fact, a GARCH model is able to explain the time-varying variance and the presence of clusters in the series of the returns. Nevertheless, it requires some constraints on both parameters and distributions of returns to obtain satisfactory results. An attractive solution is given by some mathematical models based on artificial intelligence. Indeed, the artificial neural networks, resembling the human brain, are able to make predictions of future volatility due to their ability to be self-adaptive and to be a universal approximator of any underlying nonlinear function of financial data. The aim of this thesis is to make a comparison between the forecasting capabilities of a GARCH(1,1) model and a Long Short-Term Memory network. In particular, the objective is to predict the volatility of the Dow Jones Industrial Average Index, demonstrating the superiority of the neural network with respect to the well-established GARCH model.

Contents

List of Figures

List of tables

Introduction

The uncertainty is a ubiquitous element in the current financial world, and it influences the choices of every organization, unconcerned of the size or the typology of the business. In such a situation, there is a need to measure this uncertainty, in order to predict it and make adequate plans of action. The concept of uncertainty is often associated by economists to the definition of the volatility of the returns on stock prices. The accuracy of the predictions of this statistic depends on the quality of past data and the method selected to forecast the future. Therefore, a good volatility model should be able to produce an adequate volatility forecast. But analysing and modelling financial volatility is not an easy task. The financial world is a complex entity, where plenty of factors can occur and change the evolution of things. Despite the difficulties, forecasting financial volatility is a fundamental task for many market operators. Here follow some examples. A risk manager should know the likelihood of the decline of profits in his portfolio, while an options trader would like to know the future volatility of a contract before taking a position in derivatives. Moreover, a portfolio manager may want to sell or to buy a stock on the market before it becomes too volatile, and a market maker may set the bid-ask spread broader if he knows that the market will become volatile in the future. Thus, the activity of forecasting the volatility of stock returns becomes a common concern among the practitioners of the academic world. A well-known family of models used from researchers to estimate stock volatility is the Generalized Autoregressive Conditional Heteroskedastic (GARCH) methods. These models have been proven to be able to capture most of the stylized facts of the financial data, such as the non-stationarity, the presence of cluster in volatility and the presence of autocorrelation in the squared returns.

Nevertheless, they also imposed strong restrictions on the data, regarding, for example, the shape of their distribution. These assumptions limit the accuracy of the forecasts of the model because they delimit the understanding of the dynamics of the real-world to only a few aspects.

The development of artificial neural networks has contributed to the growth of financial forecasting activity. Their ability to translate the principles of the biological processes into mathematical models makes neural networks an interesting alternative tool in data modelling techniques. As a matter of facts, ANNs can detect the underlying non-linear entity of the data and predict their future paths, without any a priori assumptions. In past years, different types of artificial neural network have been applied to different financial forecasting problem. This thesis focuses on the use of a particular type of recurrent neural networks, the long short-term memory network, to model and predict the future volatility of the Dow Jones stock index. Its capabilities will be compared with the well-established GARCH(1,1) model.

This thesis is structured in three chapters. Chapter 1 will define the concept of volatility and it will show the classical methods to model it. Then, chapter 2 will introduce the definition of the artificial neural networks and it will give an overview of their structures and a classification of their main frameworks. In the final chapter, there will be an empirical analysis based on the comparison between the predictive power of two models: the GARCH(1,1) model and the LSTM network.

1 Financial Volatility Forecast

1.1 Introduction

In recent years, increasing attention has been given to the analysis of volatility modelling and forecasting. As a measurement of the movement in stock prices, volatility is a key input to many financial applications and it becomes an important tool to assess the risk for portfolio managers, investors and other market operators (Poon & Granger, 2003). But, the modelling of volatility is a complex task, because it cannot be observed directly on the markets. Indeed, it can be measured only by looking at the extent of the movement of the stock market returns on prices. Thus, to be able to make a good forecast of future fluctuations, it is necessary using some statistical methods, that, however, requires the consideration of strong assumptions regarding the distribution of the dataset. The use of such constraints is complicated also by the essence of the financial data, which usually present volatility clusters, fat tails, nonnormality and structural breaks in the distribution of the returns. These features cannot be captured by simple classical models, such as the autoregressive moving average (ARMA) process.

This chapter presents an overview of the main estimators of the volatility, with particular attention to the heteroskedastic models: the ARCH family processes. In conclusion, it illustrates also the maximum likelihood estimation method, which represents a suitable parametric estimation process and the methodology for verifying the accuracy of a model which has been applied to the dataset.

1.2 Financial Time Series and Financial Volatility

The family of financial time series includes the series of prices of financial variables, such as stocks, currencies or commodities. The main specific of financial data is their complexity in prediction, which is a consequence of their continuously changing behaviour. To investigate how financial prices react to market news, it is necessary using statistical methods and analysing the probability distribution of past observations. Then, a good forecast requires that a suitable model is fitted to this distribution of the financial time series. A model is a substantial illustration of how next prices are computed and, if it is correct, it provides a strong similarity between simulated prices and real ones. Models are constructed by analysing time series of price data, to reap a probabilistic profile of price behaviour. As a matter of fact, a model, to be good, should be consistent with past prices. Moreover, an accurate model should be subject to a rigorous test and should be as simple as possible, according to the principle of parsimony, which establishes that models with fewer parameters to estimate are better (Taylor S. J., 2008).

As cited before, modelling financial time series is a complex task, due to some properties of financial data. In particular, stock prices have the following stylized facts (Francq & Zakoian, 2010):

- The price series are usually non-stationary and close to a random walk. A random walk is based on the assumption that price changes are random, and they wander unpredictably. This model makes impossible to predict future patterns. Therefore, stationary is a crucial element in time series analysis. There are two different definitions of stationarity. Firstly, "*a process* (X_t) *is said to be* ***strictly stationary*** *if the vectors* $(X_1, \dots, X_k)'$ *and* $(X_{1+h}, \dots, X_{k+h})'$ *have the same joint distribution, for any* $k \in \mathbb{N}$ *and any* $h \in \mathbb{Z}$*"*

(Francq & Zakoian, 2010). In other words, a process is strictly stationary if its unconditional joint probability distribution does not change when shifted in time, and so do parameters, such as mean and variance. Another weaker definition of stationarity is known as second-order stationarity, and it requires only the first moment and the autocovariance not to vary with the time. The second moment, or the variance, should only be finite.

Since stationarity is a fundamental assumption under many statistical methods, non-stationary financial data are often transformed into returns to become stationary. If p_t denotes the price of an asset at time t, the return r_t is the logarithm of the ratio between the current price and price in $t-1$:

$$\boldsymbol{r_t = log\left(\frac{p_t}{p_{t-1}}\right)} \qquad \textbf{(1.1)}$$

- The returns generally display little or no autocorrelation, suggesting a behaviour close to white noise. On the contrary, the squared returns or the absolute returns are usually strongly autocorrelated, rejecting the presence of a white noise process. A white noise is a particular type of second-order stationary process, where the mean and the autocorrelation are both equal to zero, and the variance is constant.
- The presence of volatility clustering, or persistence, is usually observed on the sample paths. Indeed, turbulent periods with high volatility are generally followed by quieter periods with low volatility. Thus, this feature of financial data is not compatible with the homoscedastic (i.e. with a constant variance) distribution for returns imposed by classical models. The volatility persistence is often associate with some phenomena, such as a slow hyperbolic

decay of the autocorrelation function, the so-called “long memory effect” of the volatility (Poon & Granger, 2005), or the mean reversion effect, that pushes backward to a normal level the volatility and it implies that the long-run forecast takes no information from the current observations (Engle & Patton, 2001). One possible consequence of mean reversion has been proven by Figlewski (1997, in Poon & Granger, 2003), who demonstrated that, even if the forecast usually improves as the frequency of data increases in respect to the forecast horizon, this phenomenon is not always true. Sometimes, if the forecast horizon is longer than ten years, using weekly or monthly data, instead of daily data, improves the estimation process.

- The graphic of the distribution of daily returns does not look like a Gaussian distribution and the normality assumption is usually rejected by classical tests. Indeed, the returns of financial time series present a fat-tailed distribution and they are said to be leptokurtic. A measure of leptokurtosis is given by the kurtosis coefficient, which is equal to 3 for the Gaussian observations. There are different impacts on current volatility from past positive and past negative values of returns. This phenomenon is called leverage and it occurs every time a negative return increases volatility by a higher amount than a positive return of the same magnitude decreases it.
- Stock prices and other financial market prices suffer from seasonality and calendar effects. The seasonal tendencies depend on the demand and the availability of an item behind a stock price, which can be not constant during the year, as, for example, the natural gas and its rise in price during winter as a consequence of its high demand. Then, the calendar effect regards each market anomaly correlated with the time of the week, month or year.

Classical models centred on the second-order stationary structure, such as an autoregressive moving average (ARMA) process, which usually represented stationary time series very well, are not enough to capture all the stylized facts just described. Sure enough, the ARMA representation, which implies a common finite variance, is incompatible with the existence of heteroskedasticity (i.e. with non-constant variance) in financial returns. Then, to get a reliable forecast of future volatility is fundamental to account of all the stylized facts. Before presenting the various approaches for volatility modelling, that takes into consideration volatility clustering of financial data, the concept of this statistical measure should be illustrated. The term "volatility" indicates the dispersion of all possible outcomes of an unforeseeable variable. In finance, volatility is calculated as the standard deviation, σ, of a set of observation and it represents the second-moment feature of the sample. The sample standard deviation can be showed by the equation 1.2:

$$\boldsymbol{\sigma = \sqrt{\frac{1}{T-1}\sum_{t=1}^{T}(r_t - \bar{r})^2}} \qquad \textbf{(1.2)}$$

Where r_t is the return on day t, and $\bar{r}$ is the mean return over T-days period. The meaning of this measure is that the higher the obtained historical volatility value is, the riskier is the security, which is not necessarily a bad thing, especially during a bull market (Taylor S. , 2004). As a matter of fact, volatility is often associated with the concept of risk. Nevertheless, there is an important difference between the two notions: volatility measures the uncertainty, which could lead to both positive or negative outcomes, whereas risk represents only the undesirable outcome. Besides, volatility gives no information about the shape of the return distribution, but it offers only a measure of its spread (Poon S.-H. , 2005).

1.3 Volatility Forecasting Models

Computing market volatility is an intricate issue because of its latent nature. In fact, differently from market prices, which are available right away, volatility has to be computed trough some mathematical calculations. Given its essence, in order to model data and to predict future volatility, a statistical model-based inference is required. This kind of statistical inference tries to select an appropriate model for inference, not knowing the one taken initially. This methodology requires though some assumptions, relating to the probability distribution of random errors, e.g. as a normal distribution, to the structural form of the relationship between variables, e.g. as a linear relationship, and to the cross-variation assumption, e.g. as errors are statistically independent.

Many methods for predicting univariate volatility exist and, in the following section, there will be a brief overview of the most popular models, which are based on the use of historical information to formulate volatility forecasts.

1.3.1 Simple Volatility Models

All the models described in this section rest on the assumption that $\sigma_{t-\tau}$, for all $\tau > 0$, is known or can be estimated at time $t-1$. The simplest historical volatility model is the random walk model, where a random noise is the only difference between volatilities of two consecutive periods and the best forecast of volatility in $t+1$ is the current volatility in t (see formula 1.3):

$$\widehat{\boldsymbol{\sigma}}_{\boldsymbol{t+1}} = \boldsymbol{\sigma}_{\boldsymbol{t}} \qquad \textbf{(1.3)}$$

In contrast, the simple moving average (MA) model makes a forecast of the sample standard error in $t+1$, based on the history of past standard deviation by keeping the size of the sample constant. Indeed, this method adds, over time, the newest observations by dropping the oldest ones, as it is showed by equation 1.4:

$$\widehat{\boldsymbol{\sigma}}_{\boldsymbol{t+1}} = \frac{\sum_{\boldsymbol{i=1}}^{\boldsymbol{\tau}} \boldsymbol{\sigma}_{\boldsymbol{t+1-i}}}{\boldsymbol{\tau}} \qquad \textbf{(1.4)}$$

where the value of τ represents the lag length to past information used. The problem with this model is that it is not able to capture the volatility clusters of financial timeseries because it weights all the observations equally. One solution is offered by the exponentially weighted moving average (EWMA): an estimation method created in the Risk Metrics framework of J.P. Morgan in 1996. The model evaluates greatly the most recent observations, by letting the weights λ exponentially declining into the past, as the formula 1.5 shows:

$$\widehat{\boldsymbol{\sigma}}_{\boldsymbol{t+1}} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{\tau}} \boldsymbol{\lambda^i \sigma_{t+1-i}} \Big/ \sum_{\boldsymbol{i=1}}^{\boldsymbol{\tau}} \boldsymbol{\lambda^i} \qquad \textbf{(1.5)}$$

where λ $(0 < \lambda < 1)$ is the decay factor, which corresponds to the parameter of the model. In this method, the λ can be seen as a weight for the last observed volatility and the smaller its value, the higher is the reaction of EWMA to recent news and changes in volatility (Ladokhin, 2009). This model has been proved to perform well and it is simple to use, but it has been forbidden by Basel Accords for the computation of VaR, because of its weights that decline to zero very quickly (Danielsson, 2011).

1.3.2 ARCH Class Conditional Volatility Models

While the volatility models described in section 1.3.1 use the constant sample standard deviations, the ARCH processes focus on the conditional variance, h_t, of returns, that is, the variance conditional on the past, and allow it to change over time leaving the unconditional variance constant. Moreover, the specific formulation of these models is able to capture many of the stylized facts presented in the previous section.

The first example of ARCH class models is the AutoRegressive Conditional Heteroscedasticity (ARCH) model, that was first proposed by Engle (1982) to capture the persistence of volatility in UK inflation. In particular, Engle proved that, even if a time series is serially uncorrelated with a mean near zero, it can differ from a white noise because of the presence of volatility clustering. This finding implies that the conditional variance of today's return, given past returns, is not constant. A common representation of ARCH(q), where q are the numbers of lags, assumes that the return series r_t are generated as follows: $r_t = \mu + \varepsilon_t$, where ε_t determines the error terms, or specifically the return residuals, with respect to a mean process μ. These error terms are seen as a product between a stochastic part z_t, which behaves as a white noise and a time-dependent standard deviation, which corresponds to the square root of the conditional variance h_t, so that:

$$\boldsymbol{\varepsilon_t = z_t\sqrt{h_t}.} \qquad \textbf{(1.6)}$$

Then, the series h_t is modelled as:

$$\boldsymbol{h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2} \qquad \textbf{(1.7)}$$

The parameters of the ARCH model, ω and α_i are unknown and they must be estimated under $q + 2$ conditions:

- $\alpha_i > 0$, for $\forall\ i = 1, \dots, q$;
- $\alpha_0 > 0$
- $\sum_{i=1}^{q} \alpha_i < 1$.

The first $q + 1$ conditions are to ensure a positive volatility forecast, while the last one is to ensure the covariance stationary with a finite variance. Then, the unconditional volatility of an ARCH(q) model, which, differently from the conditional one, depends on the entire sample, takes the following representation:

$$\boldsymbol{\sigma^2 = \frac{\alpha_0}{1 - \sum_{i=1}^{q} \alpha_i}} \qquad \textbf{(1.8)}$$

However, this restriction has not to be imposed every time, because it causes losses in parameters approximation. If the model is correctly specified is right to impose it, but all the models are faulty and allowing the parameters to be free may offer a better approximation of the true process. Moreover, if the limitation is compulsory, the estimated parameters and the resulting unconditional volatility forecast are subject to inconsistency in repeating measurements, because there is more than one combination of parameters that satisfies such constraint (Danielsson, 2011).

Following on from the construction of conditional volatility (see expression 1.7), the forecasting process of one-step-ahead can be easily compounded by knowing the past, while the forecast of the multi-step-ahead conditional variance can be estimated by supposing $E[\varepsilon_{t+\tau}^2] = \sigma_{t+\tau}^2$. Despite its innovation, ARCH model has some problem in capturing volatility, in fact, it needs a long lag length to capture the effect of historical returns on current level of volatility. In order to deal with this drawback, GARCH (p, q) has been proposed by Bollerslev (1986) and Taylor (1986). This model introduces p lags of conditional variance and it can be expressed as follows:

$$\boldsymbol{h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j h_{t-j}} \qquad \textbf{(1.9)}$$

with $\alpha_0 > 0$, $\alpha_i \geq 0$ and $\beta_j \geq 0$ (for $i = 1, \ldots, q$ and $j = 1, \ldots$,).

This equation shows two main parts: the ARCH term α_i, which measures how past errors impact on current volatility, representing the short-run persistence of shocks, that is, sudden changes on return variance; and the GARCH term β_j, which represents the contribution of past shocks to the long-run persistence, suggesting an adaptive learning mechanism. Although different methods exist to find an adequate representation of *p* and *q*, it has been proven that GARCH(1,1) is able to capture the nature of volatility in most of the financial time series (Lamoureux & Lastrapes, 1990). More recently, Miah and Rahman (2016) demonstrated that GARCH(1,1) performed better in modelling volatility of Dhaka Stock Exchange returns than other different GARCH(*p*,*q*) processes. Thus, from now on, this analysis focuses on this particular specification. The representation of the conditional variance of a GARCH(1,1) process is then showed by equation 1.10:

$$\boldsymbol{h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 h_{t-1}} \qquad \textbf{(1.10)}$$

Even in this case, the positivity constraint on conditional variance still holds. Then, $\alpha_0 > 0$, $\alpha_1 \geq 0$ and $\beta_1 \geq 0$. In order to compute the unconditional variance, the process must be covariance stationary and an additional constraint should hold: $\alpha_1 + \beta_1 < 1$. Then, the unconditional variance is given by:

$$\boldsymbol{\sigma^2 = \frac{\alpha_0}{1 - (\alpha_1 + \beta_1)}}. \qquad \textbf{(1.11)}$$

If the last condition, mentioned above, holds, the GARCH(1,1) can be interpreted as an ARCH model of infinite order. In fact, if $0 < \beta_1 < 1$ is met, there are the following steps that can be made:

$$\boldsymbol{h_t(1 - \beta_1 L) = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2} \quad \textbf{(1.12)}$$

$$\begin{aligned} \boldsymbol{h_t} &= \boldsymbol{\alpha_0(1 - \beta_1 L)^{-1} + \alpha_1(1 - \beta_1 L)^{-1}\varepsilon_{t-1}^2} \\ &= \boldsymbol{\alpha'_0 + \alpha_1(1 + \beta_1 L + \beta_1{}^2 L^2 + \cdots)\varepsilon_{t-1}^2} \\ &= \boldsymbol{\alpha'_0 + \alpha_1\varepsilon_{t-1}^2 + \alpha_1\beta_1\varepsilon_{t-1}^2 + \alpha_1\beta_1{}^2\varepsilon_{t-1}^2} \\ &\quad + \cdots \end{aligned}$$

Moreover, as pointed out by Sastra Pantula (in Bollerslev, 2007), there exists an equivalent ARMA(1,1) representation of the GARCH(1,1) model, which is given by:

$$\boldsymbol{\varepsilon_t^2 = \alpha_0 + \alpha_1\varepsilon_{t-1}^2 + \beta_1\varepsilon_{t-1}^2 - \beta_1 v_{t-1} + v_t.} \quad \textbf{(1.13)}$$

and

$$\boldsymbol{v_t = \varepsilon_t^2 - h_t = h_t(z_t{}^2 - 1)} \quad \textbf{(1.14)}$$

where $z_t \sim N(0,1)$. By merging equation 1.13 with equation 1.14, the result is:

$$\begin{aligned} \boldsymbol{h_t} &= \boldsymbol{\alpha_0 + \alpha_1\varepsilon_{t-1}^2 + \beta_1\varepsilon_{t-1}^2} \\ &\quad \boldsymbol{- \beta_1 v_{t-1}.} \end{aligned} \quad \textbf{(1.15)}$$

Although the parametrization as an ARMA process of Pantula offers a more meaningful explanation from a theoretical time series point of view, it is easier to work with the expression 1.10 (Bollerslev, 1986). However, this representation (expression 1.10) has been used and rewritten as an ARCH (∞) process by Lamoureux and Lastrapes, for explaining and measuring the

extent of the "memory" of GARCH model, that is, how long a shock to the process takes to go back to normality (Danielsson, 2011).

$$\boldsymbol{h_t = \sigma^2 + \alpha_1(v_{t-1} + (\alpha_1 + \beta_1)v_{t-2} + (\alpha_1 + \beta_1)^2 v_{t-3} + \cdots}} \quad \textbf{(1.16)}$$

The expression 1.16 shows clearly how the persistence of volatility shocks v_t depend on the sum of the GARCH parameters $(\alpha_1 + \beta_1)$. When the summation is close to zero, predictability die out very quickly. Otherwise, when $\alpha_1 + \beta_1$ becomes closer to one, the effects of past shock are stronger on current variance and it is necessary a longer period for these shocks to die out. In this latter case, the process tends to be a noncovariance-stationary, where shocks do not decay over time and the unconditional variance does not exist. This behaviour is common with daily asset returns, as a matter of fact, the measured persistence in GARCH process increases with the reduction of temporal aggregation of data (Lamoureux & Lastrapes, 1990).

Despite the above mentioned problem, the basic GARCH model presents some interesting features. Indeed, it can capture the leptokurtosis, thus the presence of fat tails in return distribution, then it is able to capture volatility clustering and nontrading periods. Moreover, it can anticipate release of public information and it has a relation with the public macroeconomic variables. However, it has some limitations: it is valid only under some constraints which impose the nonnegativity of parameters; it can not capture the leverage effect because of the independence of its response to the sign of shocks; and, lastly, it is unable to feedback from conditional variance or conditional mean. Some extensions to the basic GARCH have been proposed to overcome these limitations (Carroll & Kearney, 2009). This analysis shows only some of them, in particular the EGARCH(1,1) and the GJR-GARCH(1,1), which find a solution for the first two drawbacks. Then,

one solution to the last problem is given by the implementation of a machine learning technique, that will be presented in the next chapter of this thesis: the artificial neural network.

The exponential GARCH (EGARCH) was introduced by Nelson in 1991 and differently from the generalized ARCH, this model captures the leverage effect and, at the same time, assures that the conditional variance is always positive without imposing constraints on parameters. In the EGARCH(p,q) process, the conditional variance is an asymmetric function of lagged disturbances ε_{t-i}:

$$\boldsymbol{log(h_t) = \alpha_0 + \sum_{i=1}^{q} \alpha_i\, g(z_{t-i}) + \sum_{j=1}^{p} \beta_j\, log(h_{t-j})} \qquad \textbf{(1.17)}$$

where $g(z_t)$ takes the following specification:

$$\boldsymbol{g(z_t) = \theta z_t + \gamma[|z_t| - E(|z_t|)]} \qquad \textbf{(1.18)}$$

This construction reflects the fact that $g(z_t)$ must be a function of both the sign and the magnitude of z_t (which is defined in equation 1.6). Moreover, $g(z_t)$ is a linear function in z_t with slope coefficient equal to $\theta + \gamma$ or $\theta - \gamma$ depending on the sign of z_t, which is positive in the former case and negative in the latter. Thus, this property of $g(z_t)$, under the condition $\gamma < 0$, allows the conditional variance to respond with a higher magnitude to negative shocks, in respect to positive shocks of the same entity. In addition to its ability to capture the asymmetric effect of rises and fall of prices, the EGARCH model avoids imposition of constraint to ensure the positiveness and the covariance stationarity of the process. Nevertheless, the logarithm transformation of the conditional variance complicates the forecasting process by a mathematical point of view. For this reason, in 1993, Glosten, Jagannathan and Runkle developed an additional variation of classic GARCH model, with similar features of EGARCH process. They

developed the so called, GJR-GARCH model, which allows conditional variance to react differently to negative and positive shocks (Bollerslev, Glossary to ARCH (GARCH), 2007). The GJR-GARCH(p,q) process is expressed as follows:

$$\boldsymbol{h_t = \alpha_0 + \sum_{i=1}^{q} (\alpha_i \varepsilon_{t-i}^2 + \delta_i D_{i,t-1} \varepsilon_{t-i}^2) + \sum_{j=1}^{p} \beta_j h_{t-j}} \qquad \textbf{(1.19)}$$

with

$$\boldsymbol{D_{t-1} = \begin{cases} 1 & if\ \varepsilon_{t-i} < 0 \\ 0 & if\ \varepsilon_{t-i} > 0 \end{cases}} \qquad \textbf{(1.20)}$$

The conditional variance is non-negative if the coefficients α_0, α_i, β_j and δ_i are all nonnegative. The ability to capture the asymmetrical property is given by the indicator function D_{t-1} that assumes value 1 if ε_{t-i} is negative and 0 if ε_{t-i} is positive. Thus, positive shocks have an influence of α_i on the conditional variance, while negative shocks have an impact of $\alpha_i + \delta_i$. And so, as $\alpha_i + \delta_i > \alpha_i$, negative shocks have a bigger impact on h_t, than positive shocks have.

There exist many other types of GARCH models, but according to the aim of this analysis describing all of them would be rambling. In addition, despite the improvements of the models proposed by Nelson and Glosten (et al.) and Glosten, Jagannathan and Runkle, GARCH(1,1) has been proved to be the most appropriate for prediction in stock markets returns, and it has been depicted by Engle in 2004 the workhorse of financial applications (Carroll & Kearney, 2009).

1.4 Maximum Likelihood Estimation

Once the model is chosen and fitted to the data, the next step is to estimate model parameters. According to the nonlinear nature of the presented models, it is not possible using standard linear estimation methods, such as the ordinary least squares. Thus, the maximum likelihood will be used in this study. This type of estimation requires some prior assumptions regarding the distributions of the shocks of the model. Generally, the conditional distribution of GARCH(1,1) model is normal, and the shocks are distributed as:

$$\boldsymbol{z_t \sim N(0,1)} \tag{1.21}$$

Nevertheless, sometimes financial returns present a conditional distribution with heavy tails and one solution is the implementation of a t-GARCH. Consequently, the error term will have the following distribution:

$$\sqrt{\frac{\boldsymbol{v}}{\boldsymbol{(v-2)}}}\boldsymbol{z_t \sim t_v}. \tag{1.22}$$

In the equation 1.22, the degree of freedom, v, are estimated as an extra parameter of the GARCH model.

Considering for simplicity the normal GARCH(1,1), its the density function starts at t=2 because of the presence of lagged returns, which makes impossible defining the density function at t=1 since r_0 is unknown (see equation 1.23).

$$\boldsymbol{f(r_2)} = \\ = \frac{\boldsymbol{1}}{\sqrt{\boldsymbol{2\pi(\alpha_0 + \alpha_1 \varepsilon_1^2 + \beta_1 \sigma_1^2)}}} \boldsymbol{\exp}\left(-\frac{\boldsymbol{1}}{\boldsymbol{2}} \frac{\boldsymbol{r_2^2}}{\boldsymbol{(\alpha_0 + \alpha_1 \varepsilon_1^2 + \beta_1 \sigma_1^2)}}\right). \tag{1.23}$$

As it possible to note in equation 1.23, the existence of an extra lagged volatility term complicates the estimation. There is no estimate of σ_1, so its value can be derived as an additional parameter along with α_0, α_1 and β_1. Otherwise, another method is to set σ_1 equal to the sample variance of the returns r_t. Regarding the first solution, it may give imprecise results, but it is more recommended in the presence of a small sample size, else the second method, even if it is not theoretically correct, has impacts which are not significant if the sample is large enough. Considering the latter approach and assuming normal distribution, the log-likelihood function is the following:

$$\begin{aligned} log\mathcal{L} = &-\frac{T-1}{2} log(2\pi) \\ &-\frac{1}{2}\sum_{t=2}^{T}\left(\log(\alpha_0 + \alpha_1\varepsilon_1^2 + \beta_1\sigma_1^2)\right. \\ &\left.+\frac{r_2^2}{\alpha_0 + \alpha_1\varepsilon_1^2 + \beta_1\sigma_1^2}\right) \end{aligned} \qquad \textbf{(1.24)}$$

The maximum likelihood function estimates the parameters by numerical algorithm, which can cause some numerical issues in the process of maximization. For example, it can face difficulties in pursuing its aim if the considered likelihood function has multiple local minima or long flat area. Moreover, it can lead to numerical instability, meaning that different starting values can or cannot gives undefined results. Even if these problems are rare for smaller models like GARCH(1,1), the imposed constraint on the parameters for the covariance stationarity can bring to different right combined solutions, with serious issues as consequences (Danielsson, 2011).

Once the parameters are estimated, the next step is to produce the forecasts. Considering GARCH(1,1), a volatility forecast can be made by repeated substitutions of inputs. First, the conditional variance h_{t+1} is known and it is equal to:

$$\boldsymbol{h_{t+1} = \alpha_0 + \alpha_1 \varepsilon_t^2 + \beta_1 h_t} \qquad \textbf{(1.25)}$$

Then, given the assumption that $E[\varepsilon_{t+1}^2] = h_{t+1}$, it becomes:

$$\begin{aligned} \boldsymbol{h_{t+2}} &= \boldsymbol{\alpha_1 + \alpha_1 \varepsilon_{t+1}^2 + \beta_1 h_{t+1}} \\ &= \boldsymbol{\alpha_1 + (\alpha_1 + \beta_1) h_{t+1}}. \end{aligned} \qquad \textbf{(1.26)}$$

Similarly, as the forecast horizon increases to τ, the estimated conditional variance, after some mathematical rearrangements, becomes:

$$\boldsymbol{h_{t+\tau} = \frac{\alpha_1}{1 - (\alpha_1 + \beta_1)} + (\alpha_1 + \beta_1)^{\tau} [\alpha_1 \varepsilon_t^2 + \beta_1 h_t]}. \qquad \textbf{(1.27)}$$

1.5 Diagnosis Volatility Models

Once the model has been fitted and the parameters have been estimated, there is a need to verify that the selected model is the "best" choice among other possibilities. There are two broad types of selection method: the "in-sample" methods, which use all the dataset to choose the right forecasting model, and the "out-of-sample" methods, which ground the choice on a holdout sample, that is a sample not used in the fitting model process.

1.5.1 "In-sample" model evaluation

There are different methods to evaluate how well a model fits the data depending on whether the considered models are nested or not. In the simplest case, a model can be chosen through the implementation of the t-

test, which shows if the parameters are statistically significant different from zero. Otherwise, if two model are nested, meaning that one model is the restricted version of the other, it can be useful to employ the likelihood ratio test. Considering the larger or unrestricted model with U parameters, as M_{U}, and the smaller one with R parameters, as M_R, the likelihood ratio (LR) test can be expressed as:

$$\boldsymbol{LR = 2(\mathcal{L}_U - \mathcal{L}_R) \sim \mathcal{X}^2_{(number\ of\ restrictions)}} \qquad \textbf{(1.28)}$$

where $\mathcal{L}_U$ and $\mathcal{L}_R$ are respectively the likelihood function of the unrestricted and of the restricted models.

Another approach, to verify if a chosen model is correct and fits the data well, is to analyse its residuals, that should result independent and identically distributed (IID) normal. This check offers an assessment of how well the fitted model is able to capture the stylized facts in the dataset (Danielsson, 2011).

When two or more models are proved to fit data well, a discerning approach to compare the fit of different models on the same data is offered from the information criteria. The idea behind these methodologies is that the maximum likelihood of each model is subordinate to some penalty for the number of its parameters, which increase the model complexity. There exist two famous criteria: the Akaike information criterion (AIC) and the Bayesian information criterion (BIC), which are expressed in the following ways:

$$\boldsymbol{AIC = -2log\mathcal{L}(\varphi) + 2k} \qquad \textbf{(1.29)}$$

$$\boldsymbol{BIC = -2log\mathcal{L}(\varphi) + klog(N)} \qquad \textbf{(1.30)}$$

In both equations 1.29 and 1.30, the $log\mathcal{L}(\varphi)$ represents the maximized log likelihood function, φ all the estimated model parameters and *k* the number of parameters. In BIC there is an additional parameter *N*, which denotes the

size of the "in-sample" dataset. During the choice of a model, is important taking in account that a model with a smaller AIC and BIC offers the "best in-sample" estimation (Wennstrom, 2014).

1.5.2 "Out-of-sample" evaluation of volatility forecast

When the conditional variance models have been fitted to the data and "in-sample" evaluated, and the forecasts in the out-of-sample period has been estimated, the final step of the analysis should be an "out-of-sample" evaluation. But, evaluating the performance of the forecast is a complex task because it requires the knowledge of the realized value of the conditional volatility, that is a latent variable and it cannot be observable even ex post. For this reason, a volatility proxies is needed to proceed with the measurement of goodness of volatility forecast (Wennstrom, 2014). The proxy considered in this paper is the squared return, s_t, because, according to Danielsson (2011), *"it is a conditionally unbiased estimator of true conditional variance and is on average correct*". To understand the mathematical significance of this quote, it is important to recall the equation: $r_t = \mu + \sqrt{h_t} z_t$. From this specification, ignoring for the moment the conditional mean process μ, it is clear that the squared returns are an unbiased proxy of conditional volatility, as the expression 1.31 shows:

$$\boldsymbol{E_{t-1}[r_t^2] = E_{t-1}[h_t z_t^2] = h_t \underbrace{E_{t-1}[z_t^2]}_{1}} \qquad \textbf{(1.31)}$$

Despite it is an unbiased estimator, the use of squared returns can lead to a very imprecise forecasting because of its asymmetric distribution. Moreover, it can lead to a very low value of the R-squared, that is, a measure

that determines how much the dependent variable is explained by the independent observations. Thus, empirical findings require caution in the interpretation. Using intraday squared returns as volatility proxies can offer a solution to the presented problem, as Blair, Poon, and Taylor (2001) stated in their work, where they increase the R-squared by three to four times considering the intra-day five minute squared returns (Poon & Granger, Forecasting Volatility in Financial Markets: A Review, 2003).

A way to evaluate and compare forecast errors is through the application of an error function, which relies on the comparison between the model-based prediction on the conditional variance, $\hat{h}_t$, and one of the proxies for the conditional variance, $\hat{s}_t^2$. There exist many different evaluation measures used in the economic literature. According to the work of Poon and Granger (2003), that contains a review of 93 papers, the most common loss functions are the followings:

a. Mean Error (ME)

$$\boldsymbol{ME = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{s}_t^2 - \hat{h}_t\right)} \qquad \textbf{(1.32)}$$

This measure can be calculated as a general guide to understand if the prediction is under or over estimated. But it is not able to differentiate effects coming from errors with different signs (Brailsford & Faff, 1992)

b. Mean Square Error (MSE)

$$\boldsymbol{MSE = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{s}_t^2 - \hat{h}_t\right)^2} \qquad \textbf{(1.33)}$$

MSE tends to weight large forecast errors more harshly than other error functions do. For this reason, it is the most suitable method to determine whether a model avoids large errors.

c. Root Mean Square Error (RMSE)

$$\boldsymbol{RMSE} = \sqrt{\frac{\mathbf{1}}{\boldsymbol{N}}\sum_{\boldsymbol{i=1}}^{\boldsymbol{N}}\left(\hat{\boldsymbol{s}}_{\boldsymbol{t}}^{\mathbf{2}} - \widehat{\boldsymbol{h}}_{\boldsymbol{t}}\right)^{\mathbf{2}}} \qquad \textbf{(1.34)}$$

This measure has a broad use among academics and practitioners, even if it weights all the error at the same way, not considering their time period. One of the advantages is that, taking the square root narrows larger values of the errors towards zero, reducing the impact of outliers (Patton, 2010).

There are many others error functions, but this analysis focuses mainly on these three measures, using as proxies of conditional variance the daily square returns.

2 Artificial Neural Network

2.1 Introduction

An artificial neural network (ANN) is an information-processing system inspired to the operation of the human brain and it represents the development of a branch of the wider world of artificial intelligence (AI). From its conceiving by Alan Turing in the 30s, AI has made huge developments thanks to the increasing knowledge of the human brain and the advancing computational power of computers. The mechanism and the structure of the human brain are still too complex to be completely captured by the modern science of computers, but interesting results have been obtained trying to model the nervous system of simpler and smaller animals. Thus, from the attempts to model the biological functioning of neurons, the discipline of the artificial neural network arises as a meeting point between mathematical computation and neurobiological science. Gurney (1997) defined it as an *"interconnected assembly of simple processing elements, units or nodes, whose functionality is loosely based on the animal neuron."* In particular, it resembles the brain for its capacity to acquire knowledge from its environment using a learning process, and for its ability to store the information acquired in the interneuron connections. What makes the artificial neural network such attractive and innovative tool is its ability to change its structure in response to internal and external inputs in order to pursue a determinate task. In addition, its versatility makes possible to employ the ANN in a very wide basket of subjects, spreading from the medicine to the finance. One major application area is forecasting, indeed, thanks to its features, it offers an alternative tool to the statistical models,

presented in chapter 2, for the prediction of volatility. This chapter provides an overview of what are and how operate the artificial neural networks.

2.2 Biological Aspects

Biological neural systems are the inspiration behind artificial neural networks. For this reason, it is important to take a brief look at the organization of the human brain. The human nervous system is a three-stage process, where the centre is the neural net, that is, the brain, which continually receives electrical stimuli from the receptors, elaborates the information and then, makes suitable decision as electrical impulses that the effectors transform into discernible responses. Ramón y Cajál, in 1911, presented the idea of neurons as structural components of the brain, which simplify the knowledge of the functioning of the human nervous system. In the human cortex there are 10 billion of neurons, which are strongly interconnected between them, in fact, they originate 60 trillion of synapses, that is, connections. Usually, biological neurons are slower in operation than artificial ones, because of the massive interconnection, but they are more energetically efficient, in respect to the best computer. There exists a wide variety of neurons, Figure 2.1 shows the shape of the most common type: the pyramidal cell. As the scheme shows, the cell is the union of a

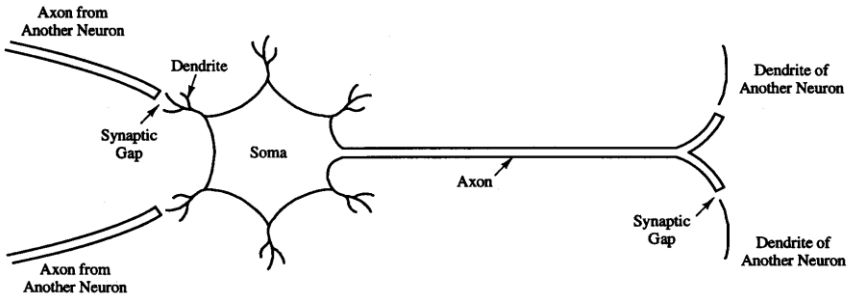


Figure 2.1: Biological Neuron

body part, called soma, and two type of filaments, the axons and the dendrites. Actually, axons are the transmission wires of the neural system and they have a great length and a few branches, while dendrites are the receptive zone and they are shorter and with more branches than axons. Neurons are linked together by chemical synapses, which are elementary units that transmit the information (Haykin, 2009). A neuron encodes the output as a series of short voltage pulses, which are known as spikes, or action potentials. These spikes originate in the soma and propagate along the axons using a domino effect. As a matter of fact, axons are constituted by a series of interrupted nodes, which fire themselves and trigger the impulse to the nearby nodes, only when they reach a certain threshold of potential. The signal pass only in one direction, because once a node has gone to its action potential, it does not fire again for a certain period of time. Then, through synapses, the impulse is transmitted to the dendrites, which operate in the same way as axons, and transmit the signal to the soma of another cell. If enough signals reach the soma and it reaches its action potential, then it fires the whole neuron and the process restarts (Bryden, 2014).

2.3 Artificial Neurons

The artificial neuron is a schematic representation of the biological one and it is called unit or node. Figure 2.2 shows the scheme of the neuron, which all the neural networks rely on. It displays a set of synapses, or connecting links, between inputs signals, $x_1, x_2, \dots, x_n$, and the neuron *k*. Each synaptic input is multiplied by a synaptic weight, w_{ki}, which is an adaptive coefficient representing the importance of the correspondent input. In the subscription w_{kj}, *k* refers to the neuron at issue, while *i* pertains to the input to which the weight concerns, with $i = (1,2,\dots,n)$. The figure also shows

a summation, that constitutes a linear combination between the input signals and their weights, and an activation function, which limits the range of the output signal to a finite value. Then, it also displays an external bias, b_k, that increases or decreases the net input of the activation function (Haykin, 2009). In mathematical terms, the neuron *k* computes the net input signal, u_k, as a linear combination output of the products between input signals and their respective signal weights:

$$\boldsymbol{u_k = \sum_{i=1}^{n} x_i w_{ki}.} \qquad \textbf{(2.1)}$$

Then, a bias v_k is added to the output u_k in order to obtain the activation potential v_k. The bias operates as an affine transformation, and, if it is equal to zero, it makes the two terms, u_k and v_k, interchangeable.

At the end, the activation function, which typically is a nonlinear function on $\mathbb{R}^n$, reduces the range of the activation potential at the closed unit interval [0,1], or alternatively, [-1.1]. Thus, it takes the following two expressions:

$$\boldsymbol{\varphi: \mathbb{R}^n \rightarrow [0, 1];} \qquad \textbf{(2.2)}$$

$$\boldsymbol{\varphi: \mathbb{R}^n \rightarrow [-1, 1].} \qquad \textbf{(2.3)}$$

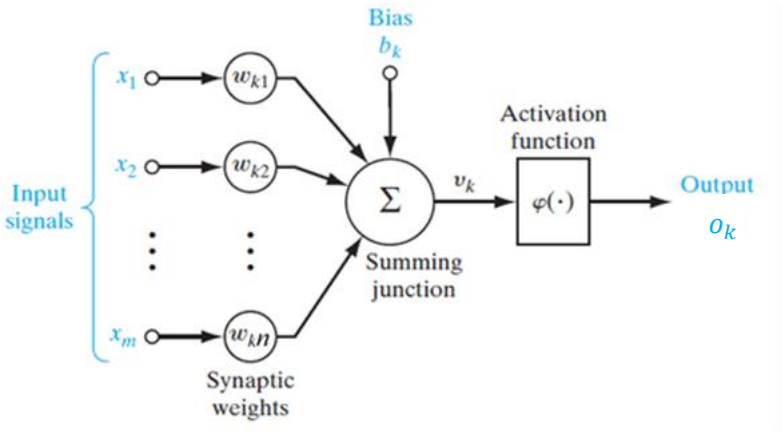


Figure 2.2: Nonlinear model of a neuron (Haykin, 2009)

Then, through the activation function, $\boldsymbol{\varphi}$, the artificial neuron produces on output signal, o_k, which takes the form of:

$$\boldsymbol{o_k = \varphi(u_k + b_k)}. \quad \textbf{(2.4)}$$

2.4 Types of Activation Functions

The aim of activation functions is to compute the output of the neuron in terms of the activation potential *v* by using some learning algorithm. There are different types of activation functions, and they can be linear or non-linear, depending on the field of the application of the neural network (Nwankpa, Ijomah, Gachagan, & Marshall, 2018). For linear models, input functions produce outputs through a linear mapping. Otherwise, nonlinear functions are useful to transform linear inputs in non-linear outputs, which then can be ready to be further processing. No rule has been defined to decide which activation function suits better the various layers. Generally, it depends from the complexity of the entire structure of the neuron. (Gallo, Artificial Neural Network in Finance Modelling, 2005). Here, there will be a short presentation of the most common functions.

2.4.1 Threshold Binary Function

The threshold activation function is characterized by Boolean signals, and it is able to take values "1" or "0", dependently if the neuron activation potential is greater or smaller than zero. Thus, the output o_k, described in Figure 2.3 is given by equation 2.5:

$$\boldsymbol{\varphi(v)} = \begin{cases} \mathbf{1} & \boldsymbol{if\ v \geq 0} \\ \mathbf{0} & \boldsymbol{if\ v < 0} \end{cases}. \quad \textbf{(2.5)}$$

A neuron with this activation function is generally referred as the MCCulloch-Pitts model, following the trailblazing work of McCulloch and Pitts (1993), who created the first and the simplest model of artificial neuron, called "threshold logic unit" (TLU). Figure 2.4 shows how the artificial neuron emulates the generation of action potential by introducing a threshold value, θ. In this situation, if the activation v exceeds or it is equal to θ the neuron emits "1" as output, while, if v is less than θ the output takes "0" as value. This model is a good schematization of the biological neuron, because it becomes active only if the activation potential reaches a determined threshold. But there are both advantages and drawbacks. Even if it is very easy and immediate to use it, it does not allow invertibility and, as a nonlinear system, the change in output does not always reflect the magnitude of changes in input (Gurney, 1997).

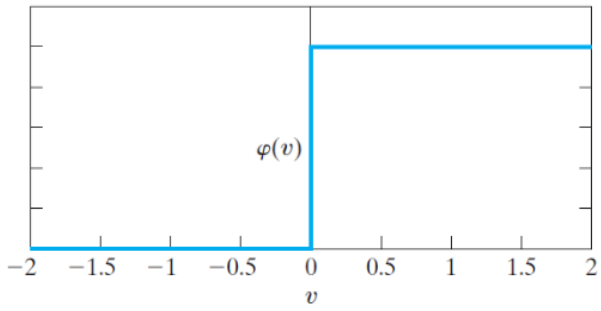


Figure 2.3: Threshold function (Haykin, 2009)

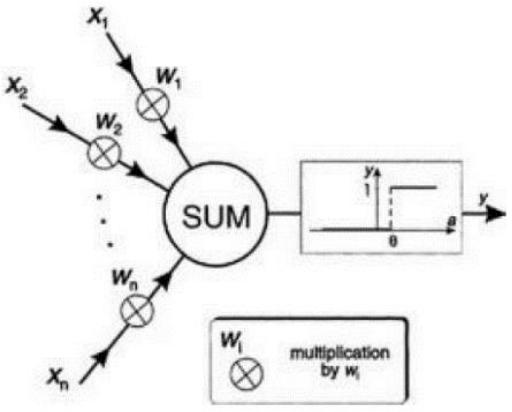


Figure 2.4: TLU (Gurney, 1997)

2.4.2 Linear Function

It is simply a linear function that can be used in either the input and output layers and it assumes the following equation:

$$\boldsymbol{\varphi(v) = hv}. \qquad \textbf{(2.6)}$$

The slope of the function is represented by the parameter *h*, which is constant. As Figure 2.5 shows, the linear function produces a linear output, which is not confined between any range. But this function is not able to deal with the complexity of data that are usually fed into a neural network. As a matter of fact, the first derivative of a linear function is equal to a constant and this causes problem with the descent gradient (that will be explain later in the chapter), which results constant. Then, another problem arises when the complexity of the neural structure increase. Indeed, even if there are many levels in the network, if all the connections are linear, the last activation function result as just a linear function of the input of the first level. So, there is a lost in information.

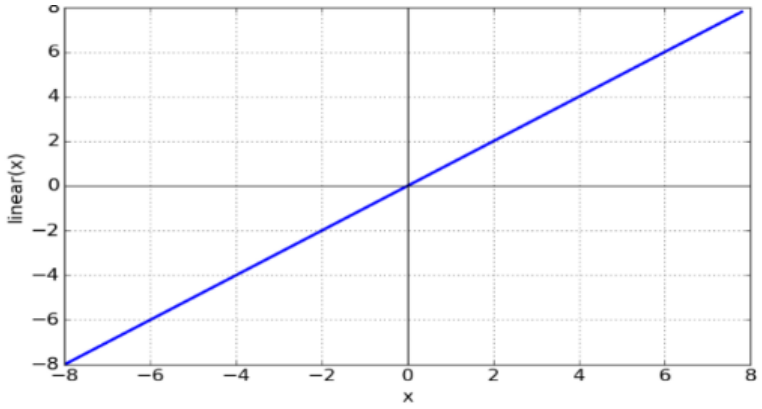


Figure 2.5: Linear Function (Sharma, 2017)

2.4.3 Sigmoid or Logistic Function

The sigmoid function, whose graph is "S"-shaped (see Figure 2.6), is one of the most used form of activation function. It is strictly increasing, and it is defined as follows:

$$\boldsymbol{\varphi(v) = \frac{1}{1 + e^{-hv}}.} \qquad \textbf{(2.7)}$$

In this case, the slope parameter, *h*, is a constant that define the steepness of the activation function and, since the function is differentiable, *h* can be found at any two points of the sigmoid curve. Moreover, if *h* tends to infinity, the sigmoid function becomes a threshold function. One of the advantages of the sigmoid function is that it can transform any inputs, assuming values ranging from minus to plus infinite, to outputs limited in the range of zero to plus one. This ability results particularly useful for models where the output is the prediction of the probability, since the probability of anything exists only between 0 and 1. But, in some case, this function is not able to capture all the dynamic characteristics of the data. In fact, towards either end of the sigmoid curve, the output values tend to

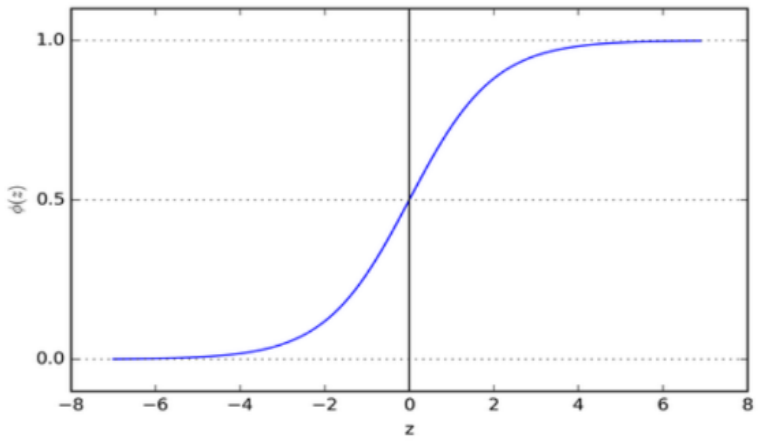


Figure 2.6: Sigmoid function (Sharma, 2017)

respond very less to changes in inputs, causing the problem of "vanishing gradients" (which is explain later in this chapter). The result of such issue is that the network learns very slowly or refuse to do it.

2.4.4 Hyperbolic Tangent Function

The hyperbolic tangent function takes the following expression:

$$\boldsymbol{\varphi(v) = \tanh(v) = \frac{e^{v} - e^{-v}}{e^{v} + e^{-v}}.} \qquad \textbf{(2.8)}$$

As shown in Figure 2.7, the tangent (green line) presents similar features to the sigmoid function (red line). Indeed, it is a nonlinear and bounded function, but, differently from the sigmoid, it is symmetric to the origin and the output range is between (-1,1). The choice between the sigmoid and the tangent function depends on the requirement of the gradient strength, because in this last case, the gradient result stronger.

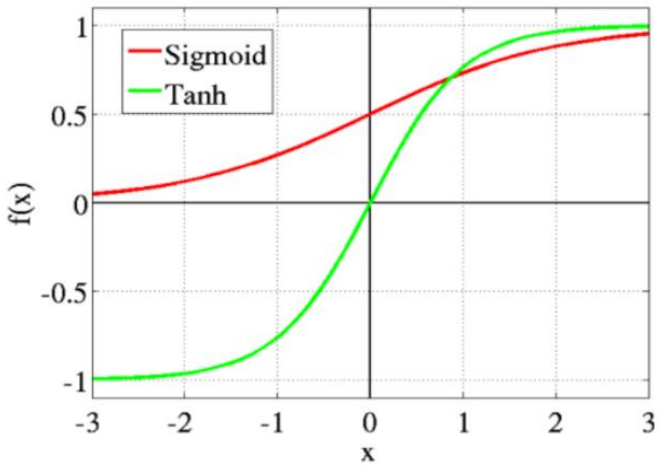


Figure 2.7: Comparison between sigmoid and tangent function (Sharma, 2017)

2.5 Network Architectures

The artificial neural network can take different structures, depending on the number of layers in which is organized. A layer is a collection of nodes, which operate together at a specific depth within a neural network and that cannot be interconnected between themselves, but only with nodes of an adjacent layer. Meaning that, the signals passes from external input nodes, organizes in the input layer, to the internal nodes of a neural network, distributed in one or multiple layers. Here, they are processed and then sent to the output nodes of the output layer. Furthermore, the structure of a network is also influenced by the learning process used to train the system, which consists in the adjustment of the synaptic weights associated to each input following the analysis of the errors committed by the network. For the moment, this subchapter focuses on the features of the different architectures. Later on the thesis, in paragraphs 2.6 and 2.7, there will be a deeper definition and an explanation of different types of learning algorithm.

2.5.1 Single-Layer Feedforward Perceptron

In the simplest form of layered network, the signals pass unidirectionally from the nodes of the input layer to the neurons of the output layer. Then, this model is said to be a feedforward single layer network and it can be outlined as in Figure 2.8. The title “single-layer” refers to the output level only, because in the input layer no computation is carried out (Haykin, 2009).

2.5.2 Multi-Layers Feedforward Preceptron

Another type of feedforward neural network is the multi-layers perceptron (MLP), which differentiates itself for the inclusion of one or more hidden layer between the input and the output layers. The term “hidden” derived from the fact that these levels do not interact directly with the external. But the aim of the hidden neurons, which are included in the hidden layers, is to intervene between the external input and the network output by adding more interactions and synapses. In this sense, the network becomes able to extract statistics from inputs of higher order and thus to offer a global perspective in spite of its local connection. This ability makes this model suitable for analysing nonlinear financial timeseries. Then, as a feedforward network, the nodes of two consecutive layers are fully connected and the connections are unidirectional, since the signal moves only in one direction from input nodes, through hidden nodes and, then to output nodes (Zhang, 1998).

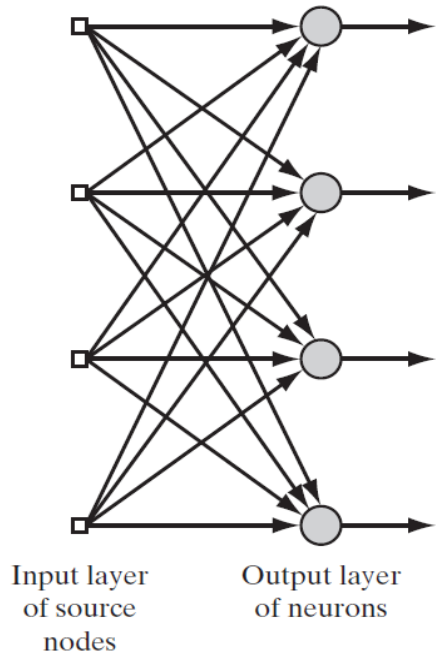


Figure 2.8: Feedforward network with a single layer of neurons (Haykin, 2009)

A common feedforward network is the model with one hidden layer (see Figure 2.9). Considering inputs $x_t = (x_{1,t}, \dots, x_{n,t})$, n the number of inputs and q the hidden units in the only hidden layer, the results are the followings:

$$\boldsymbol{h_{i,t} = \Psi\left(\gamma_{i,0} + \sum_{j=1}^{n} \gamma_{i,j} x_{j,t}\right), \qquad i = 1, \dots q} \qquad \textbf{(2.9)}$$

$$\boldsymbol{o_t = \Phi\left(\beta_0 + \sum_{i=1}^{q} \beta_i h_{i,t}\right).} \qquad \textbf{(2.10)}$$

Equation 2.9 defines the intermediate output $h_{i,t}$ of each of the q hidden layer, while equation 2.10 estimates the final output of the network o_t. In addition, Φ and Ψ are, respectively, the activation functions in the hidden and in the output layers, while $\beta_0, \beta_i, \gamma_{i,0}\ and\ \gamma_{i,j}$ are the parameters to be

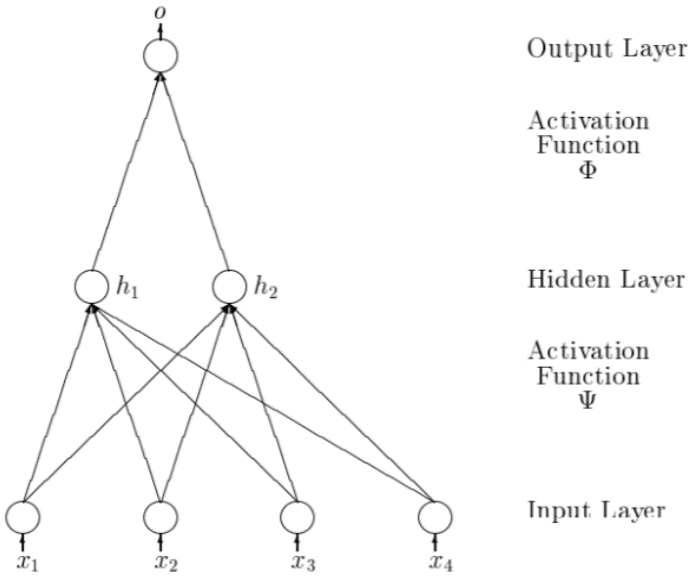


Figure 2.9: Feedforward multilayer network (Gencay & Liu, 1996)

estimated through a learning process, and that correspond to the synaptic weights (Gencay & Liu, 1996).

By increasing the number of the hidden layers, the model becomes more complex and the number of parameters increases too. If for example the comparison is made with a network with two hidden layers, the number of parameters increases by a factor $(k^* + 1)(l^* - 1) + (l^* + 1)$, where $k^*$and l^* stand for the number of neurons, respectively in the first and in the second hidden layer. But, according to McNelis (2005), more complexity means more training time and the risk to converge to a local, rather than maximum, optimum. The choice of the number of neurons in each layer depends on the problem that must be resolved. Generally, networks with fewer hidden nodes offer a better generalization ability and less problems of overfitting, but at the same time, they may not have enough power to learn and model the data. There exists no theoretical criterium to choose the optimal number, but a few approaches have been developed. The most common way is through experience, that is, via experiments. Zang et al. (1998), by reporting the previous researches of others, notice that the number of hidden neurons, that has better forecasting results, is the one equal to the number of hidden inputs. A more recent work of Dayhoff and DeLeo (2001, in McNelis, 2005), has come to the conclusion that a structure composed by one hidden layer and two or three neurons is sufficient to approximate almost any nonlinear function.

2.5.3 Recurrent Neural Network

A recurrent neural network (RNN) is similar to a feedforward network, but with the addition of feedback connections from hidden or output layers to the input layer. Depending on which layer the feedback comes from, there are two types of RNN. If the information are fed back from hidden layers, the network is a Elman RNN (Figure 2.10), while, if they are fed back from

output layers, it is a Jordan RNN (Figure 2.11). As the two figures show, in both structures, there are extra neurons in the input layer with the task to hold all the past knowledge from previous training steps. Therefore, these extra neurons are the long-term memory of the network and they are called context units (Fadda & Can, 2017). So, the innovative feature of RNNs is that it allows for both parallel and sequential computation, meaning that it can, at the same time, collect memory and evolve in time. For this reason, this type of process is more similar to the human brain than the classical feedforward neural network. Given its structure, the RNN usually use the gradient with the backpropagation through time algorithm as learning process to change its synaptic weights (see 2.5.1 chapter).

Considering again the relatively simple single hidden layer network, Elman (1990) proposed a model where the output of the hidden layer feeds back

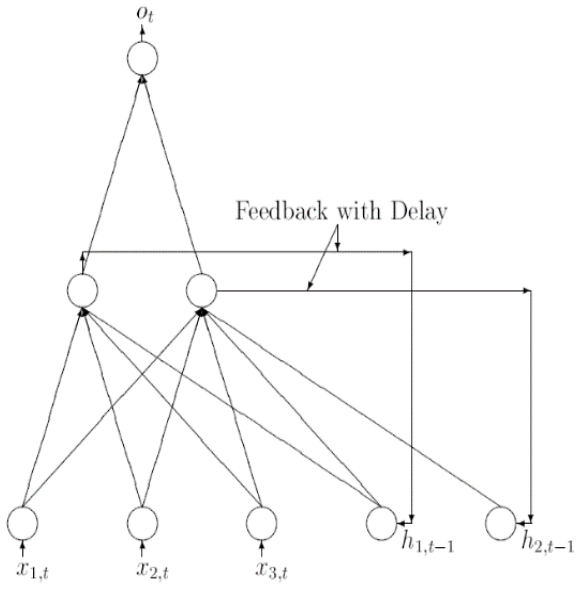


Figure 2.10:The Elman Recurrent Network (Gencay & Liu, 1996)

into the hidden layer with a time lag. The output, at time *t*, of this type of RNN can thus be represented as:

$$\boldsymbol{h_{i,t}} = \boldsymbol{\Psi}\left(\boldsymbol{\gamma_{i,0}} + \sum_{\boldsymbol{j=1}}^{\boldsymbol{n}} \boldsymbol{\gamma_{i,j}} \boldsymbol{x_{j,t}} + \sum_{\boldsymbol{l=1}}^{\boldsymbol{q}} \boldsymbol{\delta_{i,l}} \boldsymbol{h_{l,t-1}}\right) \qquad \textbf{(2.11)}$$

$$=: \boldsymbol{\psi_i}(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}, \boldsymbol{\theta});$$

$$\boldsymbol{o_t} = \boldsymbol{\Phi}\left(\boldsymbol{\beta_0} + \sum_{\boldsymbol{i=1}}^{\boldsymbol{q}} \boldsymbol{\beta_i} \boldsymbol{\psi}(\boldsymbol{x_t}, \boldsymbol{h_{t-1}}, \boldsymbol{\theta})\right) =: \boldsymbol{\phi_q}(\boldsymbol{x_t}, \boldsymbol{\theta}). \qquad \textbf{(2.12)}$$

The equation 2.11 is equal to the output of the hidden layer at time *t*, which is derived from both the inputs and the results of the hidden inputs at time $t-1$. Then, the equation 2.12 is the output of the Elman RNN. The main feature of such network is the rich dynamic behavior that creates the possibility for the network to possess a memory capacity, thanks to the

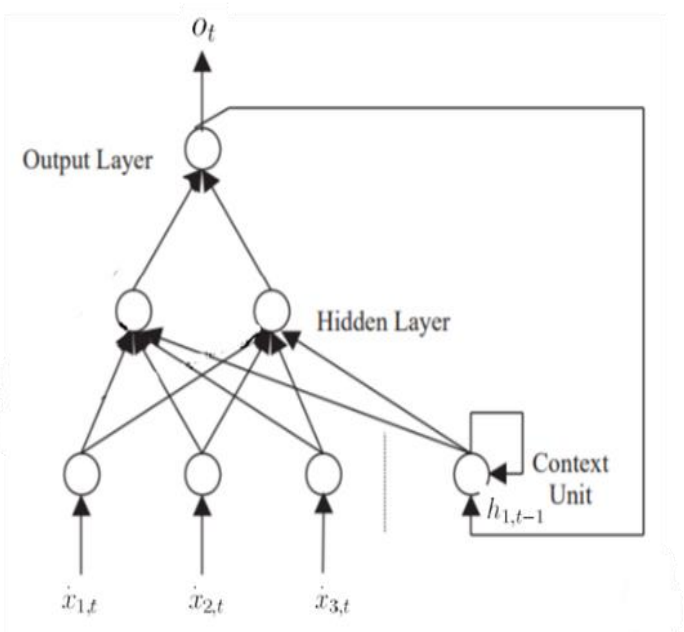


Figure 2.11:The Jordan Recurrent Network (Balkin, 1997)

presence of the context unit, and a background sensitivity, for the influence of external inputs. Indeed, since the recurrent network learning algorithms are sequential, each time period, the process is subject both to the past history of all the inputs and to new noisy data (Gencay & Liu, 1996). Despite its modelling capabilities, the RNN structure has a huge drawback that risks limiting its memory capacity. The issue is related to the optimization process of the learning algorithm and it is called vanishing or exploding gradient problem. The gradient will be described later on this chapter along the overview of the learning processes of neural networks. What is important now is that, in such situations, the network gets stuck because it enters in a chaotic regime, where the error signals shrink rapidly or grow without control. Different solutions have been proposed to handle this issue, including the modification of the structure and the use of different techniques for updating weights (Gencay & Liu, 1996).

According to the former settlement, Hochreiter and Schmidhuber (1997) introduced a new architecture: the Long Short-Term Memory (LSTM). This

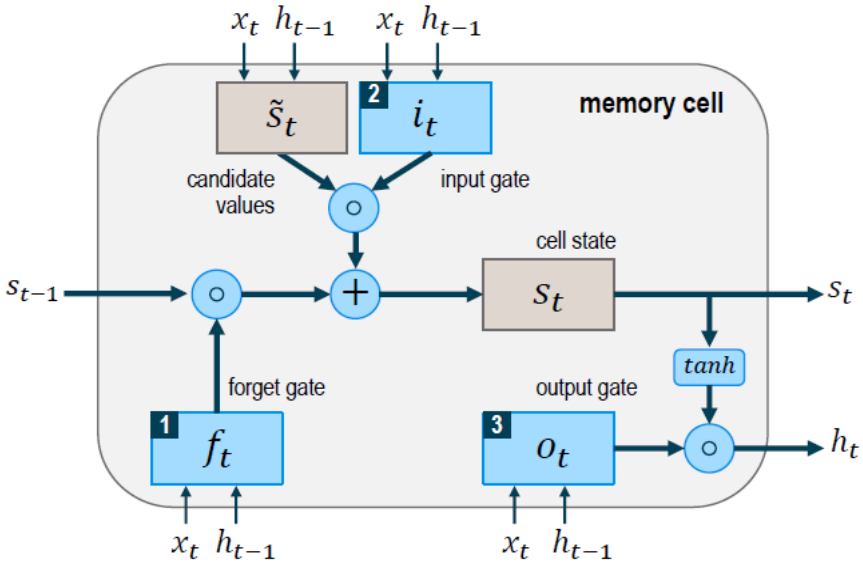


Figure 2.12: Memory Cell in LSTM network (Fischer & Krauss, 2017)

structure works in a similar way of the Elman process, as a matter of fact, it uses one input layer, one hidden layer and one output layer. But the fully self-connected hidden layer presents a complex internal processing unit, named "memory cell", with a series of corresponding regulators, usually called "gates", that supervise and manage the entering and exiting flow of information. The memory cell, which is depicted in Figure 2.12, is responsible for keeping note of all the dependencies between inputs. It is usually composed by three gates: a forget gate (f_t), an input gate (i_t), and an output gate (h_t). Each of them, at every timestep *t*, interacts with one element of the input sequence, x_t, and the output of the memory cell from the previous timestep, h_{t-1}. Specifically, the forget gate defines which information to delete from the cell state, the input gate which data to add to the memory and the output gate which results to use as output from the cell state. Each gate operates similarly to the nodes of the neural network, in fact, they act on the received signals based on the inputs and their weights, and they block or pass the information using an activation function (Fischer & Krauss, 2017). Figure 2.13 gives a representation of all the steps in which the work of the memory cell is organised and their explanations from a mathematical point of view. The first step (A) is to decide which information should be delete from the network through the forget gate. So, the activation values f_t of such gate is calculated and scaled in a range between 0 and 1 by a sigmoid function, meaning respectively "completely forget" or "completely remember". If the value is a number between them, the information is deleted or kept depending whether the produced value is nearer 0 or 1. The meaning behind the forget gate is ascribable to the fact that, sometimes, the memory cell recognizes as useless, for the assigned task, some collected information and wants to proceed by deleting them. The next step (B) is deciding which data should be added to the network's cell state, s_t. This operation happens through two phases: the use of a tanh

function which designs new candidate variables that could be added to the unit cell and the application of a sigmoid function, which defines the activation values, i_t, of the input gate. In the following step (C), there is the need to update the old state of the memory cell by combining the results of the two previous steps and by transferring them into the cell. The final stage (D) consists in creating results as the output of the memory cell. This concluding step is done, first, through the use of a sigmoid function, that define which data contained in the memory cell should be presented as the

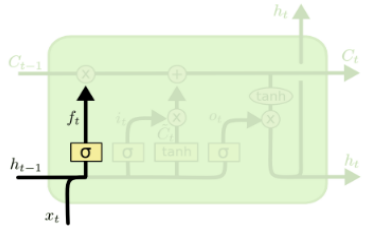
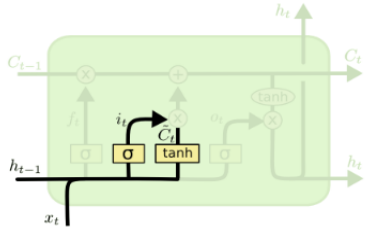
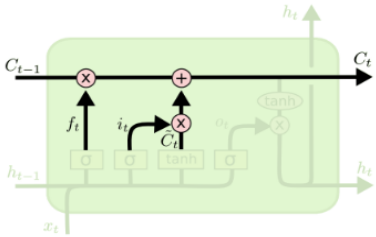
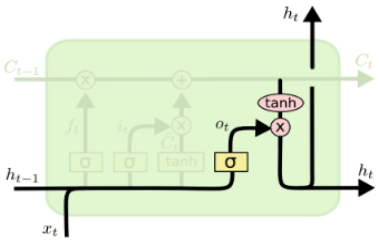
A		$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$
B		$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$ $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$
C		$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
D		$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$ $h_t = o_t * \tanh\left(C_t\right)$

Figure 2.13: Mode of operation of a memory cell in LSTM network (Olah, 2015)

output, and then, through the application of tanh function, which delimits the values of the output in a range between -1 and 1 (Olah, 2015).

2.6 Learning Processes

The human brain is neuroplastic, meaning that it can continuously change its structure and its functioning processes during the learning phases of neurons. Similarly, the artificial neural network learns through past experiences and it has the ability to change properly the weights of its input nodes. In this way, the network improves its performance on the basis of past results during the learning phase and, every time, it does better. There are two concepts correlated with the learning process: the learning paradigms and the learning algorithms. The former divides the processes into three categories: supervised, unsupervised and reinforcement learning. Otherwise, the learning algorithms classify networks depending on the way the connections between nodes are changed.

2.6.1 Supervised Learning

The supervised learning process provides to the network a training set, composed by inputs data and desired outputs. In this case, a "teacher", external to the network, helps the process to correct its responses by adjusting the weights associated with each input nodes. The aim is to minimize the error between the calculated output and real output. Once the error becomes insignificant, the weights are frozen and kept at their final values. Figure 2.14 shows how is the typical flow of an application of supervised learning to real-world problem. The first step consists in selecting the training set, which can be done following some economic logic or a "brute-force" method, whose peculiarity is to measure everything in the hope to isolate the valuable information. However, this last method can

bring to noisy data and missing values, which require a pre-processing phase. Once the dataset is chosen and pre-processed, the user must choose the learning algorithm in order to associate each input with its correct output. Then, for each pair of input and output, the network calculate the errors and it adjusts the connections weights through a learning algorithm. In a supervised process, a classical rule to rebalance weights is the Widrow-Hoff rule, or Delta Rule. This rule can be applied only to simple problems

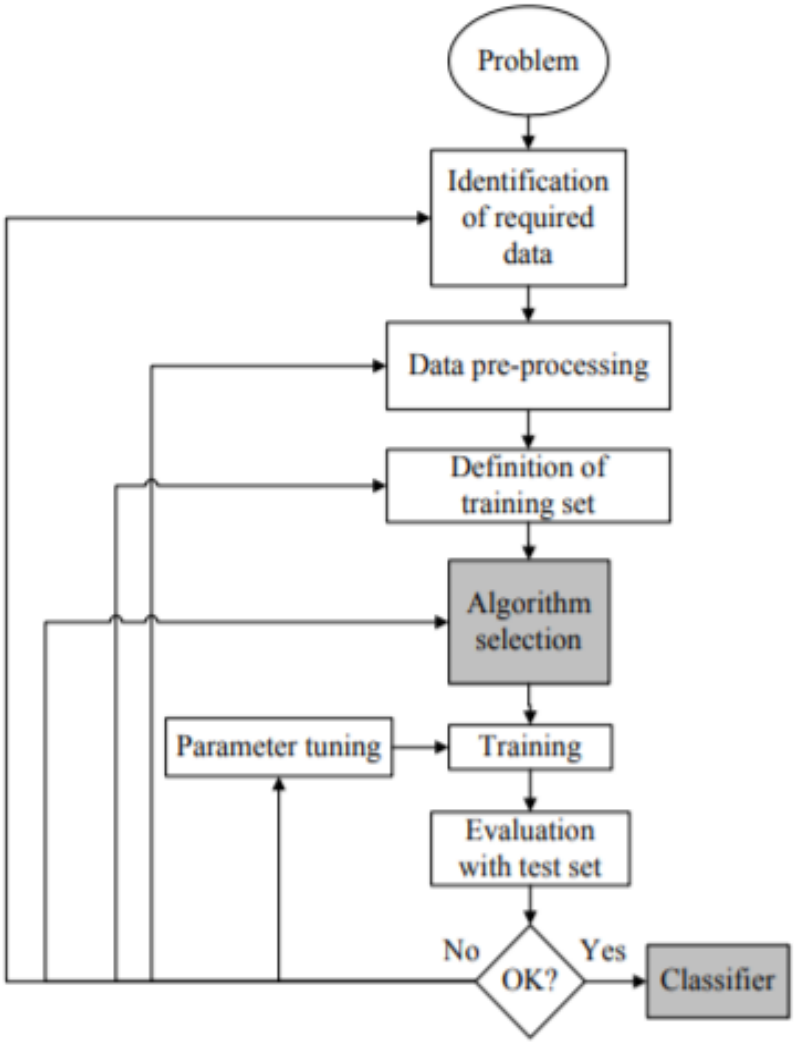


Figure 2.14: Supervised Network (Kotsiantis, 2007)

of classification because it requires long time to be computed. In a simple way, it is:

$$\Delta \boldsymbol{w_{i,j}(t)} = \boldsymbol{\alpha e_i o_j} \qquad \textbf{(2.13)}$$

where α is the learning rate and it represents a constant of proportionality; e_i is the error and it calculated as the difference between the desired and the actual output; and o_j corresponds to the output of the activation function.
As shown by equation 2.13, the only weights that are modified by this rule are the ones which contribute to the error itself. In fact, if a connection sends a zero signal, its weight remains unchanged. Otherwise, for all the other inputs the updating rule of the values of the weights are expressed by the following expression:

$$\boldsymbol{w_{i,j}(t)} = \boldsymbol{w_{i,j}(t-1)} + \Delta \boldsymbol{w_{i,j}}. \qquad \textbf{(2.14)}$$

The training phase is successful when the network is able to generalize well. The generalization consists in the ability of the model to give correct outputs to sets of inputs that it has never seen before, where the correct outputs are the ones that minimize the error between them and the real outputs (Kotsiantis, 2007). A model that is able to follow all these steps perfectly, it would be the perfect model, but, in reality, the supervised machine learning presents some issues. In fact, a model can present deficiencies in its performance, like overfitting or underfitting problems. In the first situation, the model learns "too much" from the training data, meaning that the overall error is very small, and the computed outputs become almost identical to the real ones. This state leads to some issues when new inputs are used, because the model is not able to define the principal trend of the data and thus, it is not able to generalize the pattern. On the other hand, in the second situation, the model has not "learned enough" from the training

data, which can lead to a low generalization and unreliable predictions. So, to sum up, Figure 2.15 shows that, in case of over fit *(a)*, there are small errors but high variance while, in case of under fit *(b)*, there are high errors but small variance. Then, the correct fit *(c)* corresponds to a trade-off between errors and variance.

2.6.2 Unsupervised Learning

In unsupervised learning there is no external teacher and there are only input data. The aim is to find possible common similarities in the inputs and see which ones can be generalized, and which cannot. During this type of learning process, the system uses parameters, derived exclusively from inputs, to compare and evaluate differences and similarities in order to perform a classification of the output results and adjust the connection weights (Alpaydin, 2010). One common artificial neural network that use the unsupervised learning is the self-organizing map (SOM). This network is trained using a competitive algorithm in order to reduce the dimensional representation of the input space. For an in-depth analysis of SOM see the paper of Kohonen (1990).

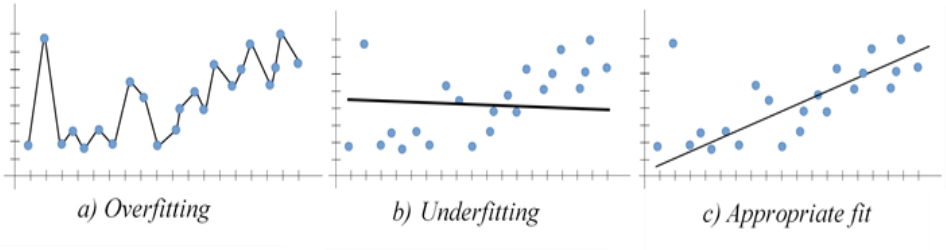


Figure 2.15: Overfitting and Underfitting Problems

2.6.3 Reinforcement Learning

Reinforcement learning is based on the relation between a software agent (i.e. a computer) and the environment. The agent takes an action in an environment, which is consequently interpreted as a reward and a representation of a state. Then, these new information are fed back into the agent, which takes new decision in the same environment, but with a broader knowledge of it. The agent can reach the best reward by taking new patterns and by learning which actions offer the best results. In the reinforcement learning what matters is the correct sequence of actions for reaching the goal. An action is considered good or not, depending on its inclusion in a good or bad policy (Alpaydin, 2010).

2.7 Optimization Process and gradient-based learning methods

The process to adjust weights in order to produce more accurate estimation consists in the following phases: formulation of the hypothesis, test of the hypothesis against reality, and redefinition of a better hypothesis. In a neural network, the set of weights associated with inputs consists in the hypothesis space of the problem. A difficult part of the work is to understand which inputs should be taken in consideration and which should be ignored. From a mathematical point of view, the optimization consists in fitting curves to data by reducing their distance from the real data-points. In other words, the network should minimize the differences between the predicted values and the values known to be true. To quantify how well the algorithm is doing in the optimization process, it is important to define the cost function (which can be called also objective, loss, or mean squared error function). A loss function maps values of one or more variables into real numbers, which

represent a "cost" related to the values. An optimization problem tries to minimize such loss function. In regression problems, it usually takes the form of the quadratic cost function, which is defined as follows:

$$\boldsymbol{C(x,w) = \frac{1}{2n}\sum_{i=1}^{n} e_i{}^2 = \frac{1}{2n}\sum_{i=1}^{n}(d_i - o_i^L)^2} \qquad \textbf{(2.15)}$$

where *n* is the total number of training examples (i.e. set of inputs associated with output), *x* is the input vector, *w* is the weight vector and e_i is the training error, which is equal to the difference between the desired output, d_i, and the actual activation output vector at the last layer, o_i^L. Note that $C(x,w)$ is non-negative and it becames closer to zero as the desired output from the network $d(x)$ becomes closer to the output activation $o^L(x)$. Optimization algorithms that try to to find weights by reducing the cost function are based on the gradient-based optimization processes. From a mathematical point of view, the gradient is a generalization of the derivative of a one-dimension function to a several dimensions function and it consists in a vector of partial derivatives of the cost function. As Figure 2.16 shows, the gradient can be represented by the slope of the loss function, which is the first derivative. In a multidimensional environment, the derivative corresponds to the tangent to the parabola function.

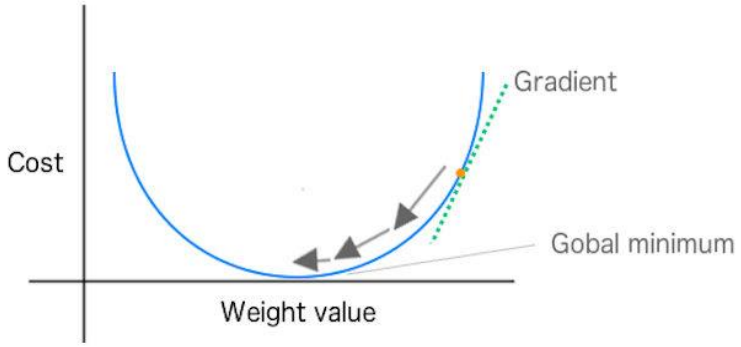


Figure 2.16: Descent Gradient

The gradient descent is a way to minimize the cost function by updating its parameters in the opposite direction of the gradient of the objective function, by a step equals to the learning rate α. To understand better how it works, it is common imagining a mountain landscape. The mountains represent the cost function. After the selection of initial values of the weights, which can be chosen by the agent or randomly, the aim is to move these initial parameters downhill as quickly as possible. The descent gradient method knows which direction is down because it can measure the change in error by changing the values of the weights. In other words, it can measure the slope of the mountains and it can give the right direction to the optimization algorithm to make to next step (see Figure 2.16).

There are three variants of gradient descent, which differentiate for the quantity of data used to compute the gradient of the cost function (Bengio & Goodfellow, 2015).

The first type to be presented is the batch (or vanilla) gradient descent (GD), which computes the gradient of the objective function using all the training dataset. Hence, to perform a single update in the value of parameters the method must compute the gradient for all the available data and this can make the process very slow. Moreover, when the unknown process generating the data is nonlinear some problem may arise. Considering the mountain landscape, with a nonlinear dataset there are several valleys (see Figure 2.17) and, in this type of environment the gradient is not able to discern if it is moving towards the lowest point of the lowest valleys, or just the lowest in a higher valley. In the latter case, the gradient can get stuck in that local minimum and the optimization process stops.

A simplification of the GD is the stochastic gradient descent (SGD). Instead of calculating the gradient of all the n training samples, each iteration estimates the gradient on the basis of a single random sample $C(x_t, w_t)$, where t is one of the possible examples among the n possibilities. Thus,

SGD allows the updates to have high fluctuations, which permit them to jump in new and potentially better local minima, avoiding the process to be stuck. On the other hand, as SGD keeps overshooting, it complicates the convergences to the exact global minimum. Nevertheless, it has been proven that if the learning rate is set to decrease slowly, SGD assumes the same convergence behavior as batch GD.

Finally, something in between these two optimization methods is the mini-batch gradient descent, which takes the best of both and performs an update for every mini-batch. A mini-batch is a collection b training samples, which splits the n training examples in n/b parts. The advantages of this method are, first, the reduction of the variance of the updates and second, the efficiency of the optimization. In literature, a common mini-batch size range between 50 and 256, depending on the dimension of the training samples (Ruder, 2016). However, vanilla mini-batch gradient descent presents a few challenges regarding the choice of the proper initial learning rate and the schedule of its reductions.

In the following section, some of the most common optimization algorithms will be presented. In particular, this dissertation starts form illustrating the evolution of three algorithms based on GD, which are the Error Backpropagation, the Gauss-Newton and the Levenberg-Marquardt and

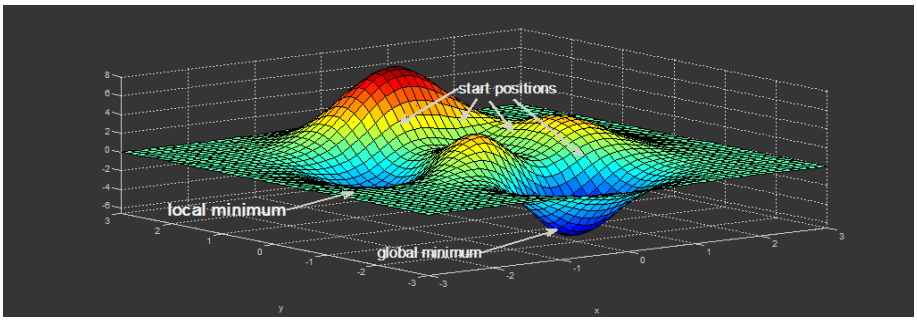


Figure 2.17: Descent Gradient in a nonlinear landscape

then it passes to the presentation of three algorithms based on SGD, that are, the momentum, the RMSProp and the ADAM.

2.7.1 Error Backpropagation Algorithm

Backpropagation algorithm made its first appereance in 1969 in a paper of Bryson and Ho. This algorithm, which is based on the steepest descent algorithm, was first ignored by researchers and it was reconsidered only in the mid-1980s, with the famaous paper of Rumelhart, Hinton and Williams, where several neural networks based on backpropation were presented. This process is an extension of the Widrow-Hoff rule and it implements the process of the gradient descent. At the heart of backpropagation there is the partial derivative of the cost function with respect to any weights in the network. In other words, it tells how quickly the cost function changes by changing the weights of different inputs. The aim of the algorithm is to compute the partial derivatives of error function with respect to any weights in the network. But, in order to apply the backpropagation algorithm, two assumptions on the cost function should be made. Firstly, it can be written as the mean of the error functions for each individual training input and, secondly, it can be seen as a function of outputs of the neural network. As it possible to see in equation 2.15, the quadratic cost function satisfies both the requirements.

From a mathematical point of view, backpropagation explains how the change in weights or in biases affects the cost function, so it computes $\partial C/\partial w_{j,n}^{l}$ and $\partial C/\partial b_{j,n}^{l}$, which are the partial derivative of cost function with respect to the weight and to the bias at the neuron j and at the layer l. Introducing an intermediate element δ_j^l, that is an error in the j^{th} neuron

and in the l^{th} layer, can help to understand better all the backpropagation process. In practice, the error δ_j^l is given by:

$$\boldsymbol{\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}} \tag{2.16}$$

where z_j^l is the weighted input to that node. The expression 2.16 is justified by the fact that a little change to the neuron's weighted input Δz_j^l changes the overal cost by an amount of $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. This change has a positive impact because it reduces the cost function, as Δz_j^l takes the opposite sign respect to $\frac{\partial C}{\partial z_j^l}$. To compute the error and the gradient of the cost function, the bakpropagation algorithm uses four fundamental equations, that splits the process in two stages: the direct or feedforward phase and the inverse or feedback phase. To understand better the meaning of the two passes, it follows a brief explanation of the four equations.

1. The error in the output layer:

$$\boldsymbol{\delta_j^L = \frac{\partial C}{\partial o_j^L}\varphi'(z_j^L)}. \tag{2.17}$$

   This expression can be splitted in two components: the first part $\frac{\partial C}{\partial o_j^L}$ explains how fast the cost is changing as function of the output activation, while the second part shows how quickly the activation function φ changes at (z_j^L). The expression can be rewritten in a simpler matrix-based formula:

$$\boldsymbol{\delta^L = \nabla_o C \odot \varphi'(z^L)} \tag{2.18}$$[1]

[1] The symbol $\odot$ represents the Hadamard product, or Schur product. Using $s \odot t$ denote the elementwise product of the two vectors.

where $\nabla_o C$ is equal to $(d - o^L)$ because the partial derivative $\frac{\partial C}{\partial o_j^L}$ of the quadratic cost function is equal to $(d - o^L)$.

2. The error δ^l in terms of the error in the next layer δ^{l+1}

$$\boldsymbol{\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \varphi'(z^l).} \quad \textbf{(2.19)}$$

The expression 2.19 can be intuitively interpreted as the backward movement trhough the activation function in layer *l* of the error. The combination of the equation 2.18 and 2.19 makes possible to compute the error term in any layer in the network.

3. The partial derivatives of the cost function with respect to any biases

$$\boldsymbol{\frac{\partial C}{\partial b_{j,k}^l} = \delta_j^l.} \quad \textbf{(2.20)}$$

This equation represents the rate of change of the cost with respect to any bias in the network.

4. The partial derivatives of the cost function with respect to any weights

$$\boldsymbol{\frac{\partial C}{\partial w_{j,n}^l} = o_k^{l-1} \delta_j^l} \quad \textbf{(2.21)}$$

where o_k^{l-1} is the output derived from the activation function at layer *l-1*. In particular, the expression 2.21 shows the rate of change of the cost with respect to any weight in the network, and it represents the gradient of the cost function (Nielsen, 2015).

So, these equations specify the way to compute the gradient of the cost function. First, the activation function for the input layer is set, then it follows a feedforward phase, where for each *l=2,3,...,L* two expressions are

computed: $z^l = w^l o^{l-1} + b^l$ and $o^l = \varphi(z^l)$. Going on, the vector of the output error is calculated, as $\delta^L = \nabla_o C \odot \varphi'(z^L)$ and then it is backpropagated for each *l=L-1, L-2, …2*, through the equation 2.19. At the end, the gradient of the cost function is found through its two partial derivatives 2.20 and 2.21, and it can be represented by the expression 2.22:

$$\boldsymbol{g = \frac{C(x,w)}{\partial w} = \left[\frac{\partial C}{\partial w_1}\ \frac{\partial C}{\partial w_2} \cdots \frac{\partial C}{\partial w_N}\right]^T.} \qquad \textbf{(2.22)}$$

Then, the update rule of the steepest descent algorithm is:

$$\boldsymbol{w_{k+1} = w_k - \alpha g_k.} \qquad \textbf{(2.23)}$$

Despite its wide use, the backpropagation is not an efficient algorithm because of its slow convergence. This problem is given by two of its features:

1. The choice of the step size of the learning rate, α, which remains constant during all the process (Figure 2.18). If it is taken very small, the optimization process reaches a good convergence, but in a very long time period. Otherwise if the step size is big, the process is time-saving, but it brings to an approximate convergence.
2. The curvature of the cost function surface, which may not be the same in all directions.

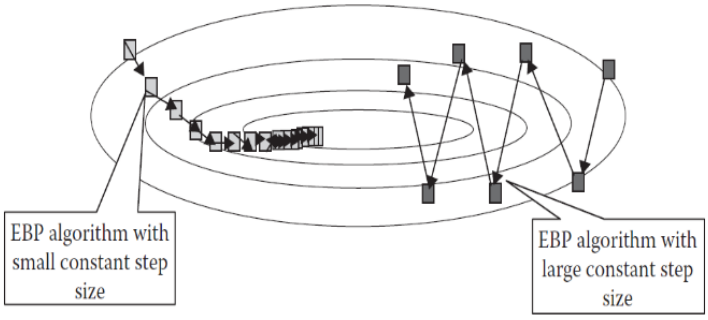


Figure 2.18:Step Size of EBP (Yu & Wilamowski, 2010)

The slow convergence problem is improved by another algorithm, the Gauss-Newton algorithm (Yu & Wilamowski, 2010).

2.7.2 Gauss-Newton Algorithm

The Gauss-Newton algorithm will be explained starting from the Newton's method for updating the weights, which is:

$$\boldsymbol{w_{k+1} = w_k - H_k^{-1} g_k} \tag{2.24}$$

where *H* is the Hessian matrix of the total error functions and it is expressed as in equation 2.25:

$$\boldsymbol{H} = \begin{bmatrix} \frac{\partial^2 C}{\partial w_1^2} & \frac{\partial^2 C}{\partial w_1 w_2} & \cdots & \frac{\partial^2 C}{\partial w_1 w_N} \\ \frac{\partial^2 C}{\partial w_2 w_1} & \frac{\partial^2 C}{\partial w_2^2} & \cdots & \frac{\partial^2 C}{\partial w_2 w_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 C}{\partial w_N w_1} & \frac{\partial^2 C}{\partial w_N w_2} & \cdots & \frac{\partial^2 C}{\partial w_N^2} \end{bmatrix}. \tag{2.25}$$

But, as calculating this matrix can be very complicated, the algorithm introduces the Jacobian matrix *J*, which has the following representation:

$$\boldsymbol{J} = \begin{bmatrix} \frac{\partial e_{1,1}}{\partial w_1} & \frac{\partial e_{1,1}}{\partial w_2} & \cdots & \frac{\partial e_{1,1}}{\partial w_N} \\ \frac{\partial e_{1,2}}{\partial w_1} & \frac{\partial e_{1,2}}{\partial w_2} & \cdots & \frac{\partial e_{1,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{1,M}}{\partial w_1} & \frac{\partial e_{1,M}}{\partial w_2} & \cdots & \frac{\partial e_{1,M}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{p,1}}{\partial w_1} & \frac{\partial e_{p,1}}{\partial w_2} & \cdots & \frac{\partial e_{p,1}}{\partial w_N} \\ \frac{\partial e_{p,2}}{\partial w_1} & \frac{\partial e_{p,2}}{\partial w_2} & \cdots & \frac{\partial e_{p,2}}{\partial w_N} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial e_{p,M}}{\partial w_1} & \frac{\partial e_{p,M}}{\partial w_2} & \cdots & \frac{\partial e_{p,M}}{\partial w_N} \end{bmatrix}. \tag{2.26}$$

By integrating the cost function and the gradient function, which are respictevely equations 2.15 and 2.22, the single elements of gradient vector can be represented as:

$$g_i = \frac{\partial C}{\partial w_i} = \frac{\partial \left(\frac{1}{2} \sum_{p=1}^{P} \sum_{m=1}^{M} {e_{p,m}}^2 \right)}{\partial w_i} = \sum_{p=1}^{P} \sum_{m=1}^{M} \left(\frac{\partial e_{p,m}}{\partial w_i} e_{p,m} \right). \quad \textbf{(2.27)}$$

Now, by combining the equation 2.27 with the Jacobian matrix (equation 2.26), the relationship between *J* and gradient vector *g* is found, and it is expressed as:

$$g = J e_{p,m}. \quad \textbf{(2.28)}$$

Moreover, the Hessian matrix can be approximeted as the product between the Jacobian matrix and the transpose Jacobian matrix. Now it is possible to define the update rule of the Gaussian-Newton algorithm as follows:

$$w_{k+1} = w_k - ({J_k}^T J_k)^{-1} J_k e_k. \quad \textbf{(2.29)}$$

One of the advantage of the Gauss-Newton algorithm is that it is able to find the proper step sizes for every directions, due to the second-order derivatives of the cost function, which evaluate the curvature of the function itself without difficulties. Thus, this ability increase the speed of convergence to the optimal estimate. But the algorithm presents also an important drawback whenever the quadratic approximation function is not reasonable. Indeed, in such situations, the algorithm diverge greatly from the true optimal, in fact the matrix $J^T J$ is not invertible (Yu & Wilamowski, 2010).

2.7.3 Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm offers a solution for the non-invertibility problem of the approximation Hessian matrix $J^T J$. To make sure it can be invertible, the algorithm introduces an additional approximation of it:

$$\boldsymbol{H = J^T J + \mu I}. \qquad \textbf{(2.30)}$$

where μ is the combination coefficient and it is always positive, while *I* is the identity matrix. This particular approximation allows the matrix to be invertible. Thus, the update rule of the algorithm is expressed by equation 2.31:

$$\boldsymbol{w_{k+1} = w_k - (J_k{}^T J_k + \mu I)^{-1} J_k e_k} \qquad \textbf{(2.31)}$$

One of the advantages of the Levenberg-Marquardt algorithm is that it is a mixture between the steepest descent and the Gauss-Newton algorithms and it takes their best features. Indeed, it inherits the stability of the former and the speed advantage of the second one. It is more robust than Gauss-Newton in dealing with complex error surface, thanks to its approximation of the Hessian matrix. Obviously, it is slower than that, but faster than the steepest descent method. As the combination of the previous two processes, the Levenberg-Marquardt algorithm switches between them during the train process. When the error surface embraces the quadratic approximation and the combination coefficient is very small, the algorithm takes the form of the Gauss-Newton and its expression becomes similar to the equation 2.29. But, when the error surface is more complex than the quadratic one and μ is large, the process slows down and starts to behave as the steepest descent algorithm (Yu & Wilamowski, 2010).

2.7.4 Stochastic Gradient Descent with Momentum

Stochastic gradient descent encounters some difficulties near the area where the surface curves become more steeply in one dimension because it oscillates across the slope of the function making very little progress towards the local optimal point. These surfaces are very common near the local optima. Introducing a fraction γ of the update vector of past observations, the so-called momentum, to the current update vector, helps to accelerate the advance and soften the fluctuations (see Figure 2.19).

So, the updated weights of the neural networks are given by:

$$\boldsymbol{w_{k+1} = w_k - v_k} \tag{2.32}$$

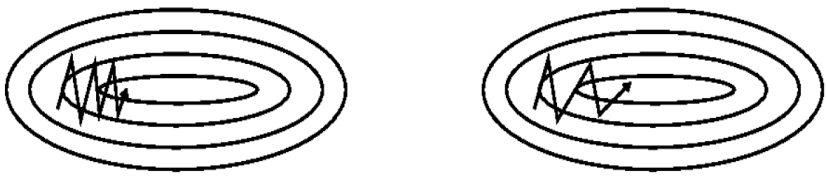
where

$$\boldsymbol{v_k = \gamma v_{k-1} + \alpha g_k} \tag{2.33}$$

The momentum becomes bigger for gradients which go in the same direction, while the parameter updates decrease as gradients change direction. This process results in a speeder convergence.

2.7.5 RMSProp Algorithm

SGD with momentum executes only a single learning rate for all the parameters. Otherwise, the Root Mean Square Propagation (RMSProp) uses



(a) SGD without momentum (b) SGD with momentum

Figure 2.19: Momentum (Bengio & Goodfellow, 2015)

different learning rates for different parameters, which can adapt to the optimized cost function. It keeps an exponentially decaying moving average of the element-wise of the squared of the values of parameters, as the expression 2.34 shows:

$$\boldsymbol{v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2} \quad \textbf{(2.34)}$$

where β_2 is the decay rate of v_k and it is usually set equal to 0.9, 0.99 or 0.999. Then, the updated parameters are computed as:

$$\boldsymbol{w_{k+1} = w_k - \frac{\alpha g_k}{\sqrt{v_k} + \epsilon}}. \quad \textbf{(2.35)}$$

The division of the equation 2.35 is element-wise and it is guaranteed to be different from zero by the addition of a small constant, ϵ. With this algorithm, the learning rate can increase, or decrease, on the basis that there are large or small gradients.

2.7.6 ADAM Algorithm

The ADAM algorithm, whose name comes from "adaptive momentum estimation", similarly to the RMSProp algorithm, computes the learning rate adapting it to the parameters. It adds an additional momentum term and it keeps an exponentially decaying moving average of the element-wise of the parameter gradients, m_t, and their squared values, v_t. The values m_t *and* v_t are the estimates of the mean and the uncentered variance of the gradient and they take the following representations:

$$\boldsymbol{m_k = \beta_1 m_{k-1} + (1-\beta_1) g_k} \quad \textbf{(2.36)}$$

$$\boldsymbol{v_k = \beta_2 v_{k-1} + (1-\beta_2) g_k^2} \quad \textbf{(2.37)}$$

where β_1 and β_2 are respectively the decay factor of the gradient and of the squared gradient. Then, ADAM uses the moving averages to update the parameters of the network and its update rule is:

$$\boldsymbol{w_{k+1} = w_k - \frac{\alpha m_k}{\sqrt{v_k} + \epsilon}.} \qquad \textbf{(2.38)}$$

Figure 2.20 shows a summary of the main features of ADAM learning algorithm and it displays also the most common default value to attribute to some of its elements.

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} =$, $\boldsymbol{r} =$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g}$ $\frac{1}{m}$ $\boldsymbol{\theta} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
t $t + 1$
Update biased first moment estimate: $\boldsymbol{s}$ $\rho^1 \boldsymbol{s} + (1 \quad \rho^1)\boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r}$ $\rho^2 \boldsymbol{r} + (1 \quad \rho^2)\boldsymbol{g}$ $\boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}}$ $\frac{\boldsymbol{s}}{1-\rho_1}$
Correct bias in second moment: $\hat{\boldsymbol{r}}$ $\frac{\boldsymbol{r}}{1-\rho_2}$
Compute update: $\Delta\boldsymbol{\theta} =$ $\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta}$ $\boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Figure 2.20: ADAM algorithm (Bengio & Goodfellow, 2015)

3 Research and Analysis

3.1 Introduction

One of the main characteristc of financial data is their nonlinear dependency on time, which cannot be captured by linear models, such as ARIMA. Such models, in fact, assume that the financial series derive by a linear process, and this is a very strong assumption to make in the real-world systems. Based on the fact that the volatility is a time-varying variable and shows clusters, a good method to model volatility is the GARCH process because it can account excess kurtosis and leverage effects. However, since GARCH is a model-based method, meaning that it imposes a model on the base of some assumptions to the dataset, it fails to capture the non-linear nature of financial data. Then, an attractive alternative is found in the artificial neural networks, since they are self-adaptive non linear data-driven methods that capture the underlying functional connection between data by learning from past examples. Their main advantage is that they do not need any prior assumptions about the model of the study, and this ability is very useful for many practical real-world based problems. Indeed, it is simpler finding data to investigate instead of guessing the underlying relationship beyond them. Moreover, this method is able to generalize starting from the existing past data. It can infer the future unseen part even if the sample observations contain noisy information. Finally, since any forecasting function assumes a relationship, which can be known or not, between past observations and the value of future output, the neural network is able to approximate any continuous function to any accuracy, since it is a universal functional approximator. On the contrary, classical models usually have some

difficulties in estimating the underlying function, because of the complexity and the non-linear nature of the real-world system (Zhang, 1998).

Despite their advantages, ANNs have also some drawbacks. In fact, building an artificial neural network forecaster is a complex task with many critical decisions to take. One should decide the appropriate architecture of the network to face the studied problem. Meaning that a decision must be taken regarding the number of layers, nodes and connections, the selection of activation functions, the training algorithm, the size of training and test sets and the methods of performance measures (Zhang, 1998).

In the remainder of this chapter, there is a brief reference to the main previous researches about applying the artificial neural network to the forecasting task of financial time series. Then, later on this chapter, there is the development of the empirical project: it starts from analysing the forecasting abilities of a GARCH(1,1) model, considering both a normal and a t-student distribution, and it concludes the analysis with a series of experiments where a LSTM is applied, first, to the returns and, later, to the raw data.

3.2 Literature Review

The first application of artificial neural networks in forecasting problems dates back to 1964 when Hu tried to predict the weather using an adaptive linear network. At that time, there was a limited knowledge of the training algorithms and the work had not a huge success. Later, when Rumelhart et al. (1986) discovered the backpropagation algorithm, the application of ANN for forecasting started to spread among researchers. Attention must be given to the first work of Lapedes and Farber (1987), who concluded that neural networks are a suitable tool to predict nonlinear time series. Indeed, they created a feedforward neural network which was able to forecast

accurately the evolution of two chaotic time series, that is, series that show an apparently random behaviour. In 1995 Cottrell et al. focus on the implementation of NN to forecast the real-world time series (Zhang, 1998). Afterwards, many researchers were proposed to investigate this problem and huge efforts have been made to create literature over the comparison between ANNs and statistical models in time series forecasting. For an in-depth analysis, Zhang (1998) offered a summary on the trailblazing works in ANN forecasting and later, Atsalakis & Valavanis (2008) created a survey over 100 scientific articles regarding the use of ANN to predict the walk of a single or many stocks, in both developed and emerging markets.

Following the reputation in financial forecasting of neural networks, some more recent works have focused on the task of volatility prediction capabilities of NNs were first explored by Ormoneit & Neuneier (1996), that tried, for the first time, to predict the volatility of the DAX index using density estimating neural networks. Then, one year later, Donaldson and Kamstra (1997) provided pieces of evidence of the superiority of artificial neural networks over the classical ARCH-type model in both in-sample and out-of-sample forecasts for four international stock indices: S&P500, NIKKEI, FTSE and TSEC. Monfared and Enke (2014) studied how is the performance of NNs in predicting future volatility of the NASDAQ Composite Index during four different economic cycles, from 1997-2011. They concluded that it successfully outperforms all the other econometric models, especially during tumultuous periods, such as the financial crisis of 2008. However, it is useless implementing artificial intelligence in low volatility periods, because of its unnecessary complexity (Monfared & Enke, 2014).

Besides the advantages proposed by the traditional neural networks in the stock market forecast activity, these innovative tools may also fail to predict the behaviour of a stock because of the problem of the presence and the

selection of local optimal as weights. For this reason, in recent years big attention has been given to a new branch of artificial neural networks, the recurrent neural network and, in particular, to the Long Short Term Memory networks. There exist some works over the application of LSTM to the prediction of future returns of a stock or an index. Some examples are the seminar proposed by Hansson (2017) on the superiority of the LSTM networks over the classical model ARMA(1,1)-GJRGARCH(1,1), and the work of Namin Siami et al. (2018), which shown the superior ability to predict stock prices of LSTM against an ARIMA model. Despite these researches, very few attempts have been made over the volatility forecastability of LSTM neural networks. Then, this work would like to contribute to enlarging the literature about the forecasting process of the volatility of a stock index using neural networks. In particular, this thesis aims to prove the superiority of a LSTM network over a well-established GARCH(1,1) process in this field of application.

3.3 Empirical project

This thesis aims to select an effective forecast model for the volatility of a univariate financial time series, specifically a stock index. For this reason, the variable of interest is the Dow Jones Industrial Average (DJIA), a stock market index measuring the stock performance of 30 companies listed on stock exchanges in the USA. The closing prices were downloaded using Bloomberg in the period ranging from 13/07/2015 to 11/07/2019 (see Figure 3.1). The time series seems to be non-stationary with considerable variations in the level, in fact, the lowest value is 15˙660 (11-02-2016) while the highest value is 27˙071 (11-07-2019). The constant mean is excluded, and it is possible to notice an increasing trend. Since the dataset represents a financial time series with a non-stationary behaviour, the data are

transformed in returns on prices with the application of the log transformation and the first difference (see Figure 3.2).

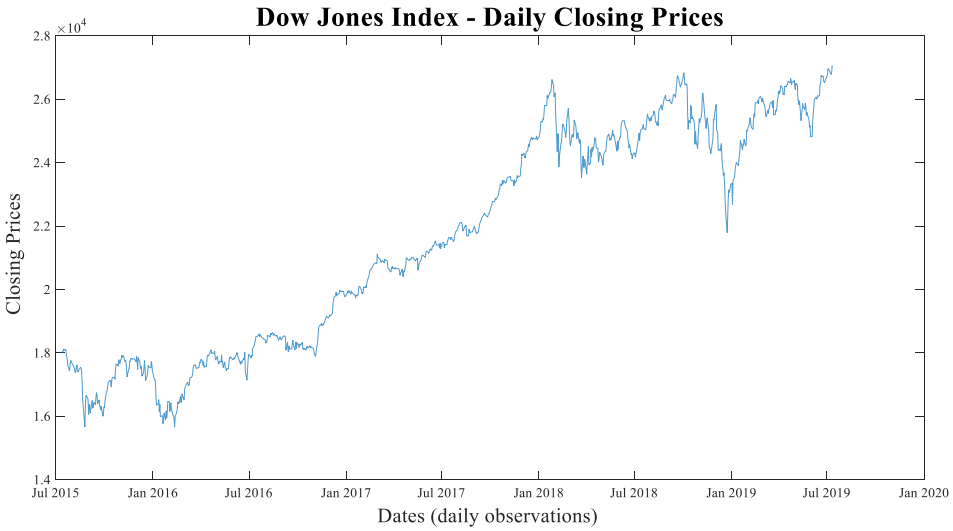


Figure 3.1: Closing Prices (data processing in MATLAB)

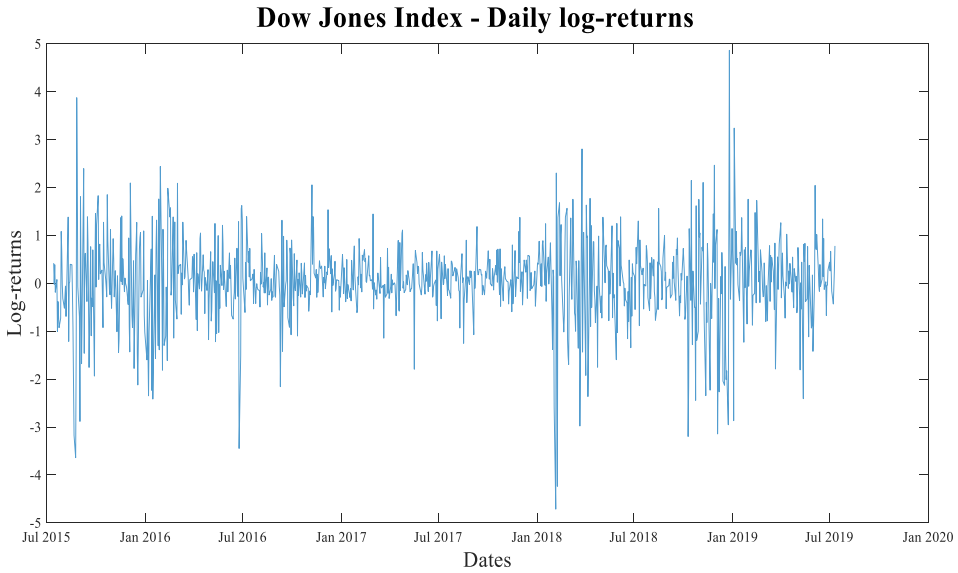


Figure 3.2: Returns (data processing in MATLAB)

Table 1 reports the main statistics of the returns on the Dow Jones Industrial Average index:

Mean	Variance	Standard Deviation	Maximum	Minimum	Skewness	Kurtosis
0.0406878	0.751896	0.8671194	4.8643312	-4.7142779	-0.5248143	7.2913674

Table 3.1: Main Statistics of returns (Data processing in MATLAB and Excel)

As it is possible to notice, the distribution of returns is leptokurtic (see the kurtosis value), slightly asymmetric (see skewness value) and heteroskedastic (see the cluster in Figure 3.2). The presence of heteroskedasticity can be also verified by the Engle's ARCH test, which gives a test statistic equal to 89.6337, with a p-value smaller than 0.05. Hence, the null hypothesis, assessing that a series of residuals exhibit no conditional heteroscedasticity, is rejected. Such behaviour is confirmed also from the presence of graphical clusters in the representation of the squared returns (see Figure 3.3), which are used as a proxy of the volatility of the

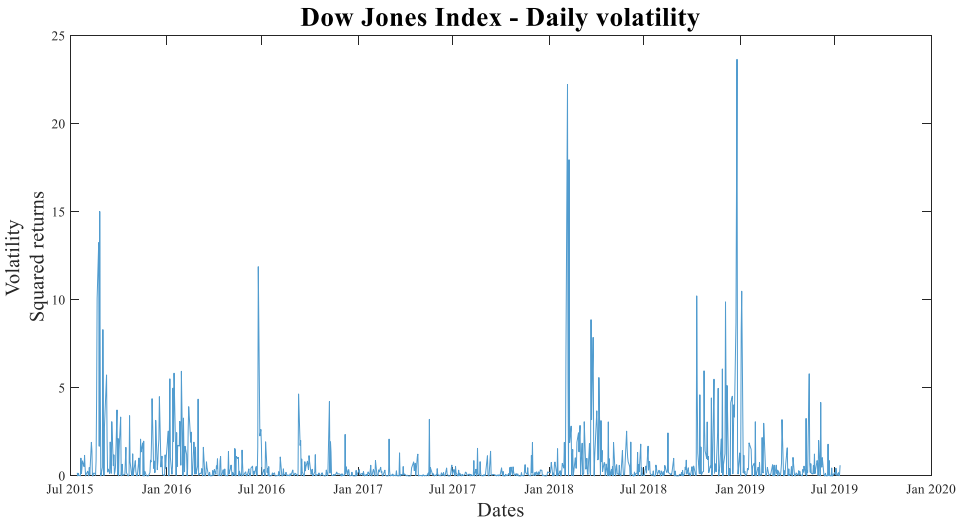


Figure 3.3: Squared returns (data processing in MATLAB)

index. Moreover, checking the plot of the autocorrelation function of returns and squared returns, it can be seen that, in the former case (see Figure 3.4), most of all lags are enclosed in the confidence boundaries, while in the latter case (see Figure 3.5), the first 14 lags are out of the confidence boundaries. Meaning that the Dow Jones index has no autocorrelation in the returns but shows strong evidence of autocorrelation in the squared returns. Such behaviours are aligned with key features of financial time series described in the first chapter. All these considerations are the typical features that characterize the family of ARCH/GARCH processes.

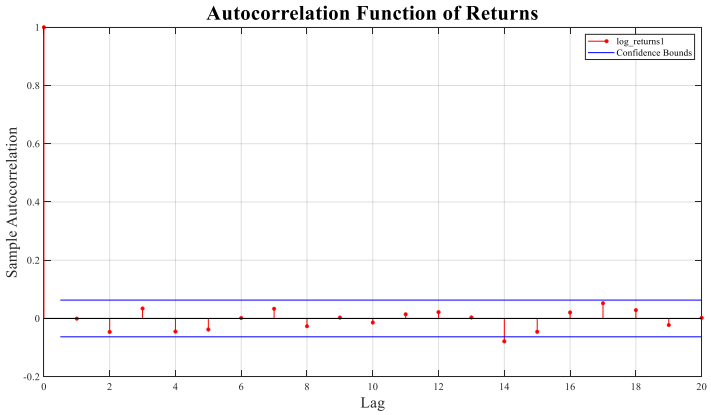


Figure 3.4: ACF Returns (data processing in MATLAB)

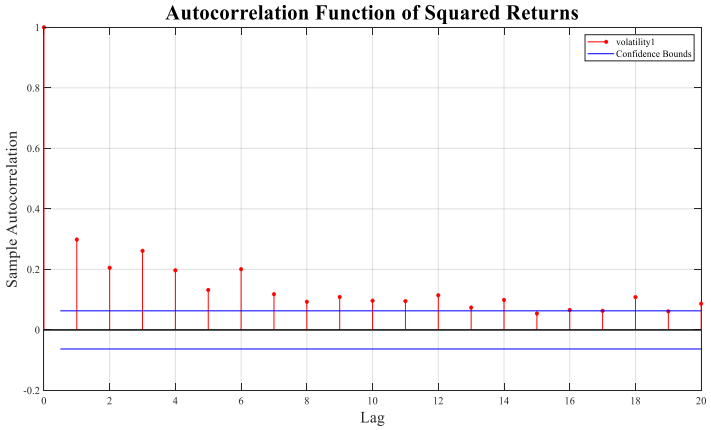


Figure 3.5: ACF Squared Returns (data processing in MATLAB)

3.3.1 Application of GARCH

In order to apply the GARCH model to the returns, the dataset is divided into two sub-samples:

- The in-sample set: from 14/07/2015 to 10/07/2018 (754 observations);
- The out-of-sample set: from 11/07/2018 to 11/07/2019 (252 observations).

Then, a GARCH(1,1) model is applied to the in-sample dataset, as the following equation states:

$$\boldsymbol{h_t = \alpha_0 + \alpha_1 \varepsilon_{t-1}^2 + \beta_1 h_{t-1}} \qquad \textbf{(3.1)}$$

with $\alpha_0 > 0$, $\alpha_i \geq 0$ and $\beta_j \geq 0$ (for $i = 1, \dots, q$ and $j = 1, \dots$,).

In order to make a complete analysis, two modelling frameworks with different error distributions are investigated. First, a GARCH(1,1) model with normal error distribution is examined, and then, its results are compared to the outcomes of a GARCH(1,1) with a t-student distribution of the errors, from both in-sample and out-of-sample perspective.
Starting from the GARCH(1,1) with normally distributed errors, the estimated parameters are the ones presented in table 2.

Parameter	Value	StandardError	TStatistic	PValue
Constant	0.046379	0.0074769	6.203	5.5411E-10
GARCH{1}	0.71949	0.030998	23.211	3.5078E-119
ARCH{1}	0.21159	0.025286	8.3681	5.855E-17

Table 3.2: GARCH(1,1) with normal error distribution - (Data processing in MATLAB and Excel)

It is possible to notice that all the parameters are greater than zero, since $\alpha_0 = 0.046$, $\alpha_1 = 0.212$ and $\beta_1 = 0.719$. Moreover, as $\alpha_1 + \beta_1 = 0.931$, the process is covariance stationery because the summation is smaller than 1, but it presents a high persistence of volatility since the value is very close to 1. Then, the effects of past shocks are stronger on current variance than recent ones, and it is necessary a longer period for these shocks to die out. This behaviour is aligned with the literature since the dataset used is on a daily basis. Similar results are obtained fitting a GARCH(1,1) model with a t-student distribution of the errors, i.e. with $\sqrt{\frac{v}{(v-2)}} z_t \sim t_v$, as shown by table 3:

Parameter	Value	StandardError	TStatistic	PValue
Constant	0.024613	0.0093412	2.6349	0.0084173
GARCH{1}	0.77217	0.043063	17.931	6.7615E-72
ARCH{1}	0.22261	0.053123	4.1906	0.000027825
DoF	4.4017	0.7999	5.5028	3.7385E-08

Table 3.3: GARCH(1,1) with t-student error distribution - (Data processing in MATLAB and Excel)

Even in this case, all the parameters are greater than zero, in fact: $\alpha_0 = 0.025$, $\alpha_1 = 0.223$ and $\beta_1 = 0.772$, with $\alpha_1 + \beta_1 = 0.995$ <1.

If the two GARCH models are evaluated in terms of AIC and BIC, the one with the t-student distribution of errors appears to be the slightly more suitable, as table 4 illustrates:

Error Distribution	AIC	BIC
Normal	1.60E+03	1.62E+03
T-Student	1.54E+03	1.56E+03

Table 3.4: In-sample analysis of GARCH models (Data processing in MATLAB and Excel)

However, if the two models are investigated in out-of-sample terms, through the consideration of the error functions, it is possible to conclude that: since the mean error functions are close to zero there is no under or over estimation in both models; then, since both MSE e RMSE are smaller for GARCH with normal distribution than GARCH with t-distributed errors, the choice of the best model redounds on the former (see table 5).

Error Distribution	ME	MSE	RMSE
Normal	0.059546	4.6572	2.158
T-Student	-0.096904	4.7741	2.185

Table 3.5: Out-of-sample analysis of GARCH models (Data processing in MATLAB and Excel)

The values of the error functions displayed in table 5 are calculated using the squared returns as a proxy of conditional volatility. Since the focus of this dissertation is a comparison between the out-of-sample forecasting powers of a GARCH(1,1) model and a LSTM network, it will be given more prominence to the results of the out-of-sample analysis. Indeed, from now

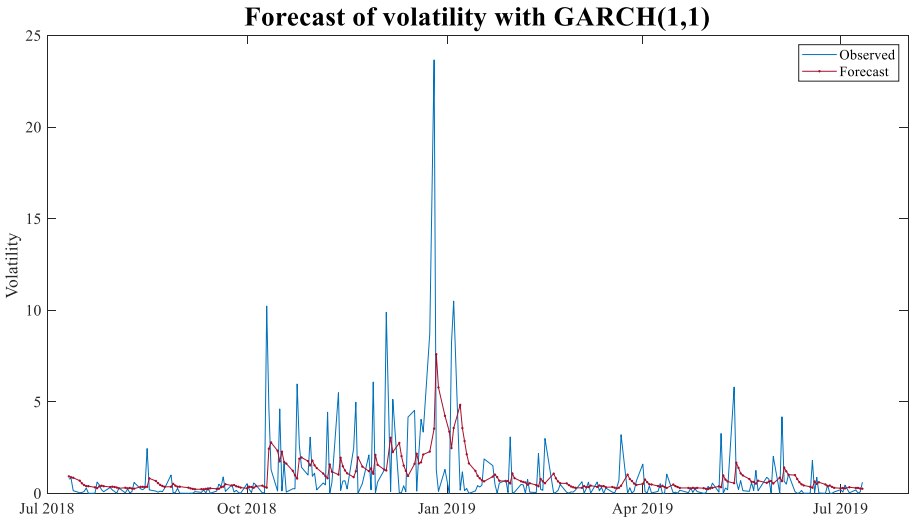


Figure 3.6: Volatility forecast with GARCH(1,1) – (Data processing in MATLAB)

on, only the GARCH(1,1) with normal distribution will be taken into consideration. Figure 3.6 displays the prediction capability of such model. A GARCH (1,1) can offers a good approximate view on which periods are characterized by high volatility and which ones not. Thus, this model offers a valid solution for market operators who want to hedge themselves from the risk of turmoiled period. Now, the question is if the new mathematical models can beat the forecast abilities of the GARCH in a case of a univariate time-series. In the following section a LSTM network will be implemented in order to verify if can give a better estimate of volatility.

3.3.2 Application of Long Short-Term Memory Network

The artificial neural network considered in this thesis is a particular type of recurrent neural network, that is, the Long Short-Term Memory. LSTM is one of the most advanced technique of machine learning for pursuing tasks of sequence learning, such as time series prediction. Xiong et al (2015) have demonstrated the power of LSTM in predicting the volatility of S&P500 considering the google trends, but there were no previous attempts to use this method to predict the volatility for a univariate index using only past observations of the time series. This dissertation tries to fill this void and apply a LSTM network to forecast volatility of the Dow Jones Industrial Average index, using only the historical returns, and then it will compare the results with the outcomes of the GARCH model previously presented.
A methodology to design a neural network forecasting model was proposed by Kaastra & Boyd (1996), which established eight steps that may be revisiting a few times. All the steps are displayed in the following table:

Design a neural network:
Step 1: Variable selection
Step 2: Data collection
Step 3: Data pre-processing
Step 4: Training and testing sets
Step 5: Neural network paradigm: - Number of hidden layers - Number of hidden neurons - Number of output neuron - Transfer function
Step 6: Evaluation criteria
Step 7: Neural network training - Number of training iterations - Learning rate
Step 8: Implementation

Table 3.6: How to design a Neural Network (Kaastra & Boyd, 1996)

The following paragraphs will present a deeper analysis of each step adapted to the construction of a Long Short-Term Memory network.

Variable selection, Data Collection and Software

For the empirical application, the data used are the daily log-returns on closing prices of the Dow Jones Industrial Average index. The time-series of 1006 observations were originated from the log-transformation of the daily closing prices downloaded from Bloomberg. The time period for returns ranges from 14/07/2015 to 11/07/2019. Data preparation and handling are entirely conducted in MATLAB R2019b, using the Deep Learning Toolbox™, which provides a framework for designing and implementing deep neural networks with pre-trained models and algorithms

(The MathWorks, 2019). The LSTM network is trained on the following CPU: Intel® Core™ i7-8550U CPU @ 1.80GHz.

Data Pre-processing

This phase includes the analysis and the transformation of the input and the output variables to minimize noises, to detect trends and to highlight important relationship among data. It is often useful to normalize the raw data in order to achieve higher results during the optimization phase of the objective function. There exist many different methods involved in the normalization process, including:

- Rescaling or min-max normalization:

$$x' = \frac{x - min(x)}{max(x) - min(x)} \quad \textbf{(3.2)}$$

- Mean normalization:

$$x' = \frac{x - average(x)}{max(x) - min(x)} \quad \textbf{(3.3)}$$

- Standardization or Z-score Normalization:

$$x' = \frac{x - \bar{x}}{\sigma} \quad \textbf{(3.4)}$$

The first two methods consist in rescaling the range of features in the range of *[0, 1]* or *[−1, 1]*. Otherwise, the latter makes the values of the variable have zero-mean and unit-variance. This last method is often used in machine learning techniques because it helps the stochastic gradient descent to improve the speed of the convergence to the optimal. Then, all the daily returns are transformed using the formula in equation 3.4.

Training and testing set

The dataset needs to be divided into training and testing sets, in the same way as they were divided for the GARCH analysis into in-sample and out-of-sample datasets. So, there are:

- The training set: from 14/07/2015 to 13/07/2018 (754 observations);
- The testing set: from 11/07/2018 to 11/07/ 2019 (252 observations).

The training set is the part of data on which the network learns and trains itself. In other words, this set is used to make all the computations, such as the definition of the gradient and the update of weights and biases. This part is divided in turn in two sub-sections: the actual training set and the validation set (see Figure 3.7). This latter set of data is useful to understand if the network is overfitted to the original values. Overfitting leads to a very poor generalization capability and to bad forecasting performances in terms of error functions, and, for this reason, it should be avoided. The training set is so divided:

- Training sub-section: from 14/07/2015 to 13/02/2018 (653 observations);
- Validation sub-section: from 14/02/2018 to 13/07/2018 (101 observations).

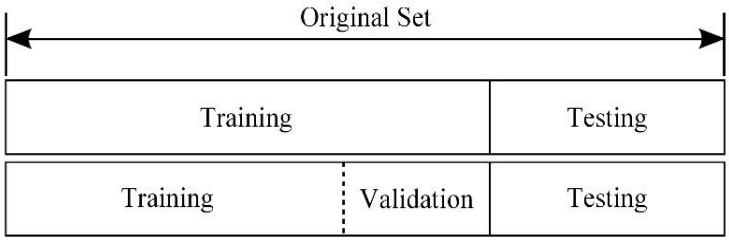


Figure 3.7: Dataset Division for NNs

Whereas, the testing set is used to make a final check on the predicted out-of-sample values by measuring the network performance in the forecasting ability.

Neural network paradigm

There are several different ways to build a neural network, depending on the choice of the number of layers, the number of hidden neurons and output neurons and the transfer function. The neural network chosen for this paper is composed of:

- 1 input layer
- 1 LSTM layer
- 1 fully interconnected layer
- 1 output layer

Focusing on the long short-term memory layer, it is constituted by 250 hidden units. This number corresponds to the amount of information remembered among time steps and it has been selected by trial and error since there is no 'magic' formula for deciding the optimum value. During the choice one fact has been taken into consideration: a very large number of hidden units leads to an overfit in the training data. Another fundamental operation in building a neural network is the selection of activation functions. The LSTM network has two different functions, one for the state and one for the gate. The state activation function, representing the process that updates the cell and the hidden state, has been set as a tangent function by default. Whereas, the gate activation function, which is the process applied to the gates, has been set as a sigmoid.

In addition to the LSTM layer, the network has also a fully connected layer, which is useful to multiply the input by a weight matrix and adds a bias

vector, and an output layer, which contains only one output neuron, since neural networks with multiple output neurons produce inferior results compared to networks with a single output.

Evaluation Criteria

To evaluate the performance of both the training and the validation set, the error function used is the Root Mean Squared Error. Furthermore, the generalization abilities of the network are evaluated using the 3 error functions previously implemented for the GARCH model: ME, MSE and RMSE. Even in this case, the functions are built as a comparison between the volatility forecast, derived from the LSTM network, and the proxy of volatility, represented by the squared of returns on closing prices. The volatility forecasts do not represent the outputs of the network, but they can be easily derived taking the squared of the output returns.

Neural Network Training

Training a neural network involves the process to find the set of optimal weights between the neurons, whose combination determines the global minimum of the objective function, that is, the mean square error function. The final set of weights usually gives a good generalization, unless the network is overfitted. The network uses a stochastic gradient descent training algorithm and it adjusts the weights moving down itself along the steepest slope of the error surface. The algorithm selected in this paper is the ADAM algorithm (see paragraph 2.7.6 of this thesis), whose features, displayed in table 7, have been selected by trials and errors.

TrainingOptionsADAM with properties:	
MaxEpochs	150
MiniBatchSize	128
InitialLearnRate	0.007
LearnRateSchedule	'piecewise'
LearnRateDropPeriod	100
LearnRateDropFactor	0.8
GradientDecayFactor	0.9
SquaredGradientDecayFactor	0.999
Epsilon	1.00E-08
ValidationFrequency	10
ValidationPatience	Inf
Shuffle	'never'
ExecutionEnvironment	'cpu'
Plots	'training-progress'

Table 3.7: Training Options of LSTM (MATLAB)

Starting from 2000 epochs, after many attempts, this value was decreased to 150. A possible explanation of this reduction is given by the graphic (see Figure 3.8) proposed by Kaastra et al. (1996). As the number of epochs increases, the error function of the training set becomes closer to zero, meaning that the network is able to learn perfectly how the training set

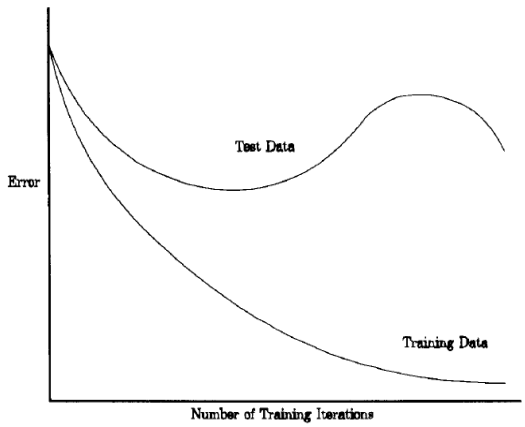


Figure 3.8: Possible neural network training and testing set errors (Kaastra & Boyd, 1996)

behaves, but, at the same time, the error function of the testing data, after an initial reduction, starts to increase. This behaviour makes difficult for the network to understand the general path and the trend of the unseen part of the dataset. So, it makes the prediction less reliable.

Implementation and results

Starting from the structure presented in the previous section, with 150 epochs and without any validation accuracy limit, which is a parameter that avoids the overestimation in the testing phase, the algorithm is modified until it reaches satisfactory results. After each change, the network was trained three times, and only the best result in terms of error functions is here displayed.

Experiment 1

Considering the basic design presented in table 7, the network is trained until epoch 150. As Figure 3.9 displays, the training ability to learn the path of the training set becomes finer as the epochs go on. However, the

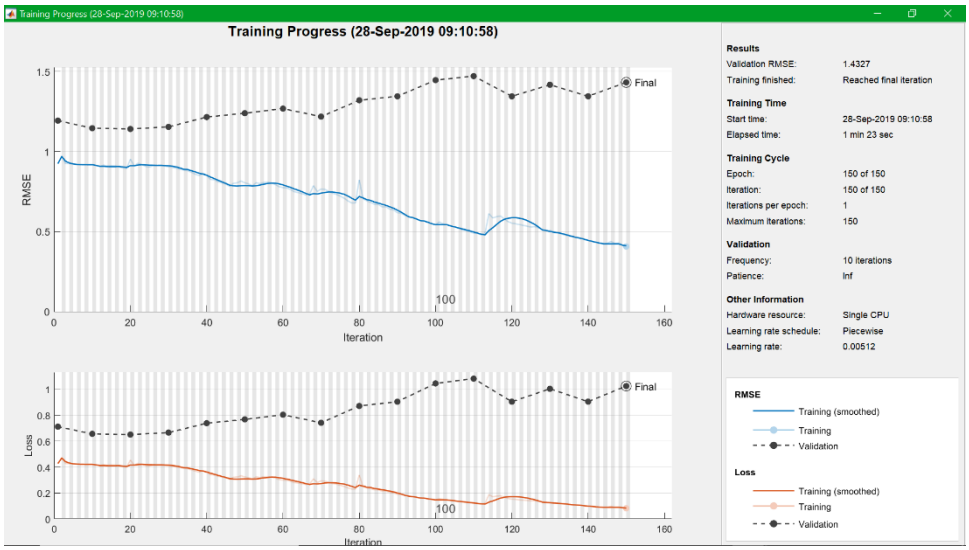


Figure 3.9: Training progress -LSTM basic (MATLAB)

capability to generalize what learned in the training set to a never-seen-before subset of data (i.e. the validation set) gets worst as the epochs increases. In other words, the network suffers of overestimation issues, that lead to a very poor forecasting performance in comparison to a GARCH(1,1) model. The presence of overestimation in LSTM network is also confirmed by the value taken by the mean error function displayed in table 8, which is much bigger than the one taken by GARCH. In addition, since both MSE and RMSE functions are smaller for LSTM than for GARCH process, it is possible to conclude that, in this case, the network cannot beat the heteroskedastic model in forecasting performance.

Model	ME	MSE	RMSE
GARCH(1,1)	0.059546	4.6572	2.158
LSTM - basic	0.34637	4.9924	2.2344

Table 3.8: Error functions GARCH and LSTM-basic (Data processing in MATLAB and Excel)

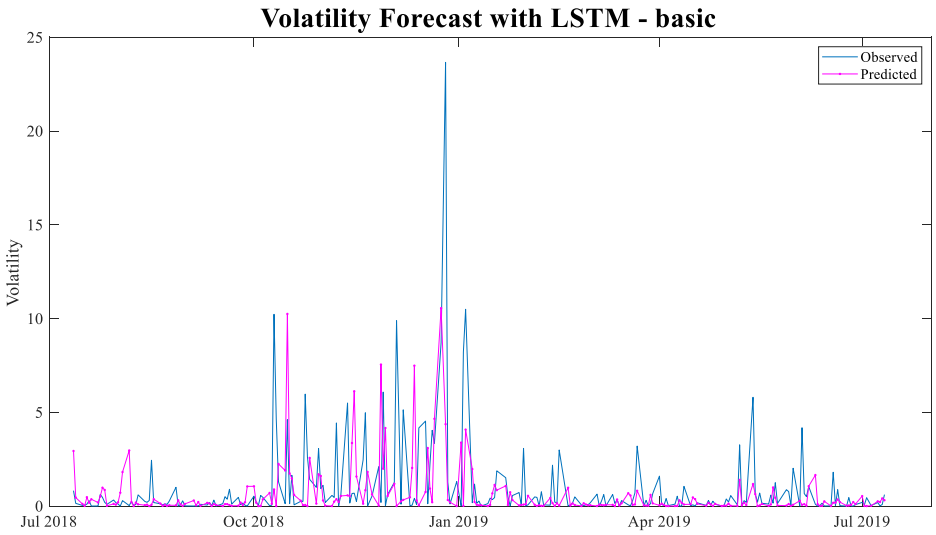


Figure 3.10: Volatility Forecast LSTM-basic (Data processing in MATLAB)

Figure 3.10 illustrates all the above-mentioned problems. Indeed, even if the network has learnt to predict spikes in volatility, it cannot understand when these spikes will exactly verify in the future, with the consequence to increase the values of the error functions. However, it seems to distinguish between period of high or low fluctuation in prices.

Experiment 2

In order to overcome this issue, a validation patience of 5 is imposed on the validation set. Meaning that if the loss on the validation set is greater or equal to the previously smallest lost for 5 times, the network training stops. Figure 3.11 displays that the network stops at 120 epochs because it has met the validation criterion. Even this early stopping has occurred, the performance of the network decreases (see table 9). The mean errors increase, and the same considerations can be done for the MSE and the RMSE.

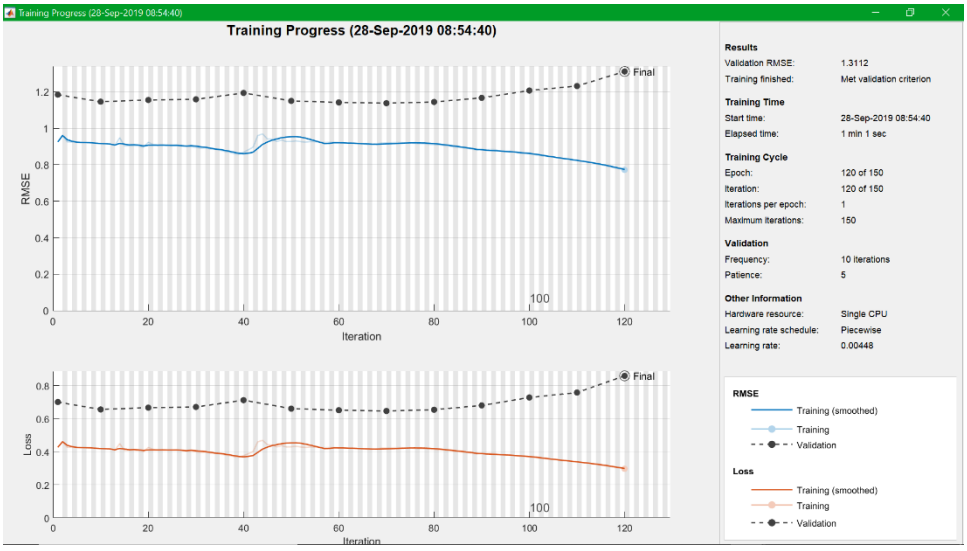


Figure 3.11: Training progress - LSTM with validation patience (MATLAB)

Model	ME	MSE	RMSE
GARCH(1,1)	0.059546	4.6572	2.158
LSTM - val 5	0.70922	5.4343	2.3312

Table 3.9: Error functions GARCH and LSTM-validation patience 5 (MATLAB and Excel)

As a matter of fact, Figure 3.12 shows that, in the forecasting process, the network is able to understand the general path of volatility, but it seems to underestimate the spikes during periods of high turmoil. Probably this is a consequence of the early stop in the learning phase.

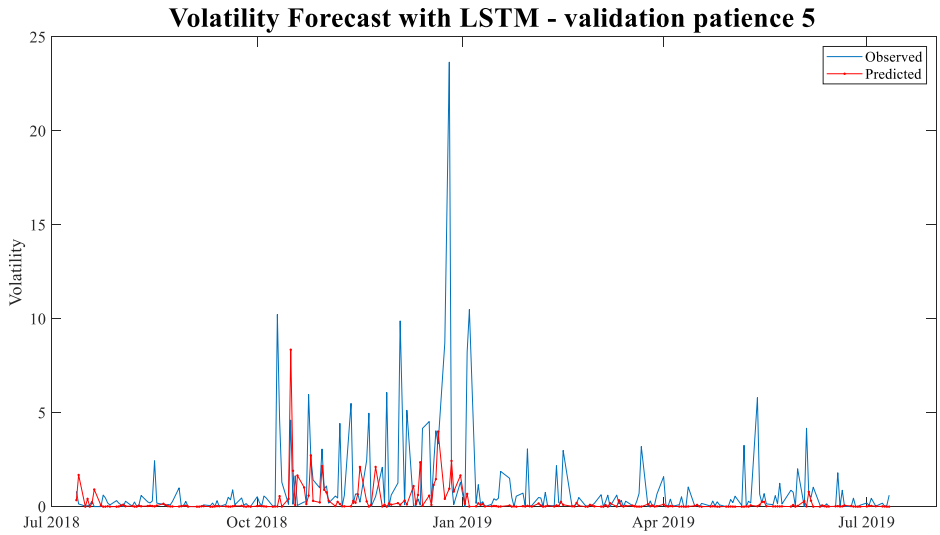


Figure 3.12: Volatility Forecast LSTM with validation patience (MATLAB)

Experiment 3

In this experiment, the learning rate is reduced from 0.007 to 0.003 and the validation patience is left to 5. The rationale behind this reduction is that there is a need for caution in order to achieve good results before the early stop for validation patience occurs. Thus, a small initial learning rate makes the network adjust slowly and carefully, avoiding the possibility of overshooting. The results achieved with this structure are satisfactory one over three times. As table 10 shows, both the MSE and the RMSE are smaller than the ones obtained from a GARCH(1,1) process. But, the mean error is still high, meaning that this process is again vulnerable to overestimation issues.

Model	ME	MSE	RMSE
GARCH(1,1)	0.059546	4.6572	2.158
LSTM - slr/val5	0.60422	4.4838	2.1175

Table 3.10: Error functions GARCH and LSTM with smaller init. learning rate (MATLAB and Excel)

Figure 3.13 displays that the network is stopped after 70 epochs for having reached the validation condition. However, the loss function of both the testing set and the validation set are passable, since, in the former case, the value is below 0.4, while in the latter case, the value has not reached 0.8 (differently from the previous two experiments). Moreover, the out-of-sample estimates of the test sample seem to be able to capture both the periods of low and high volatility, with a satisfactory precision during the latter ones (see Figure 3.14).

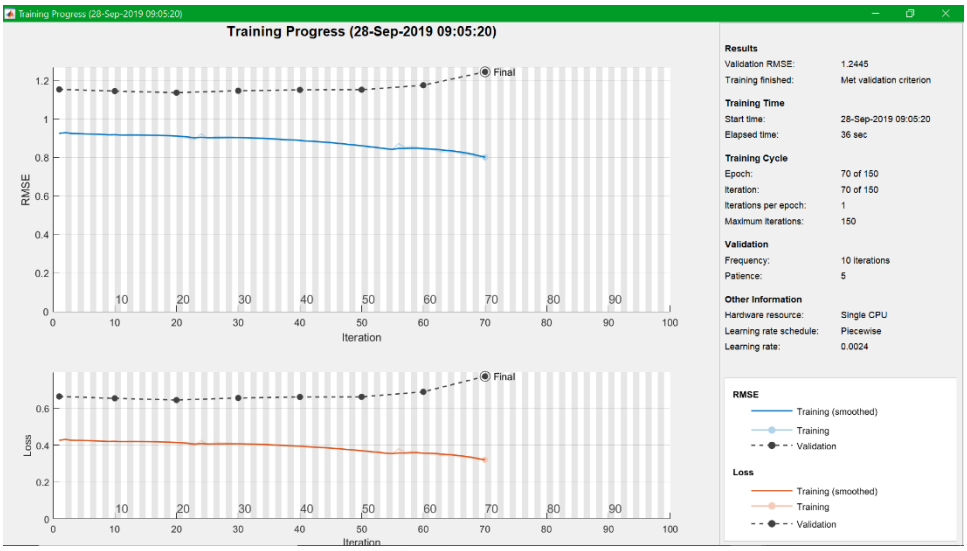


Figure 3.13: Training progress - LSTM with smaller init. learning rate (MATLAB)

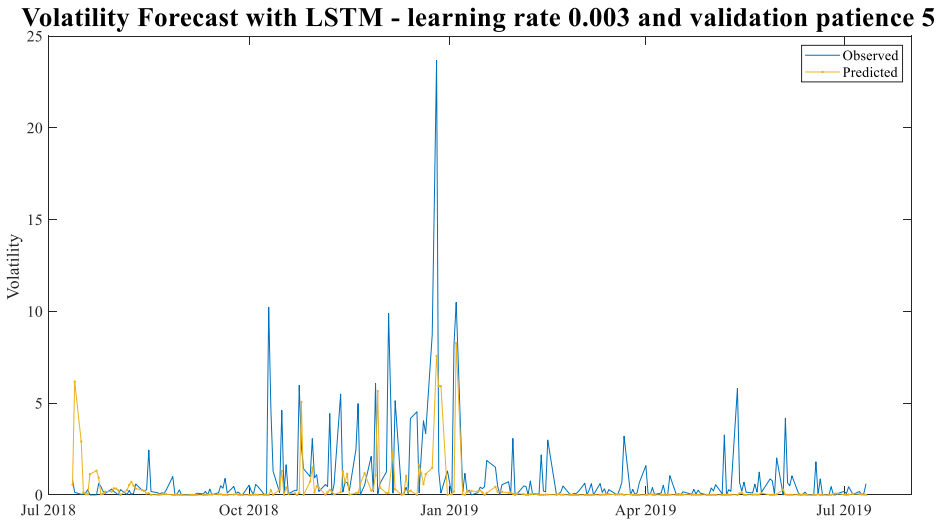


Figure 3.14: Volatility Forecast LSTM with smaller init. learning rate (MATLAB)

Despite the good results obtained in experiment 3, the network continues to show a problem, which is common also to other experiments: the validation error function is increasing while the training error function is decreasing. A personal interpretation of this issue dates back to the choice of the input dataset. It is difficult not to occur in overfitting problem in dealing with returns since their path tends to oscillate in a recurring way around the mean value, which is very close to zero. This behaviour makes difficult to understand the main direction of data and the generalization process since they do not show a trend in their path. Following this line of reasoning, the following experiment considers as input variables the closing prices of Dow Jones Industrial Average stock index, instead of the returns on prices, while all the other conditions are set equal to the ones presented in table 7, with the only exception in the number of epochs, which is fixed to 70.

Experiment 4

Even in this environment, the network is run three times, resulting, 2 over 3 times, in a better model than GARCH(1,1). Here, there are presented the results of the best experiment. As table 11 shows, both the MSE and the RMSE of this network beats the results of a GARCH(1,1). Moreover, the loss functions of both the training set and the validation set decrease exponentially to zero (see Figure 3.15). So, the network is able to learn very fast during the training phase and it can generalize better than any other model. The out-of-sample performances are shown in Figure 3.16. The model seems to understand the general path of volatility, but it underestimates the predicted values during periods of high volatility. Due to this last observation, there is no need to proceed with the reduction of the learning rate, since a smaller value of it can only reduce the power of the learning phase during periods of turmoil.

Model	ME	MSE	RMSE
GARCH(1,1)	0.059546	4.6572	2.158
LSTM - CP	0.66123	4.5058	2.1227

Table 3.11: Error functions GARCH and LSTM with CP as input (MATLAB and Excel)

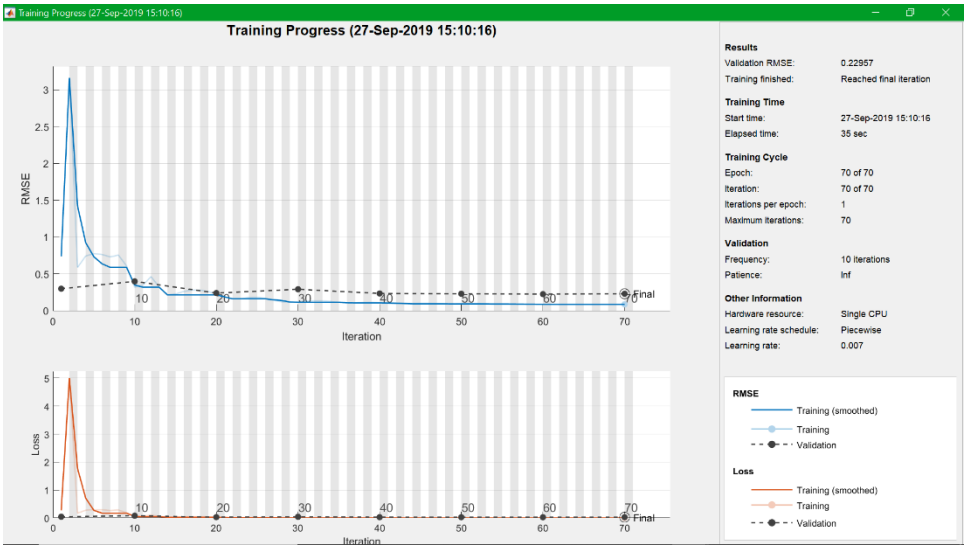


Figure 3.15: Training progress - LSTM with CP as input (MATLAB)

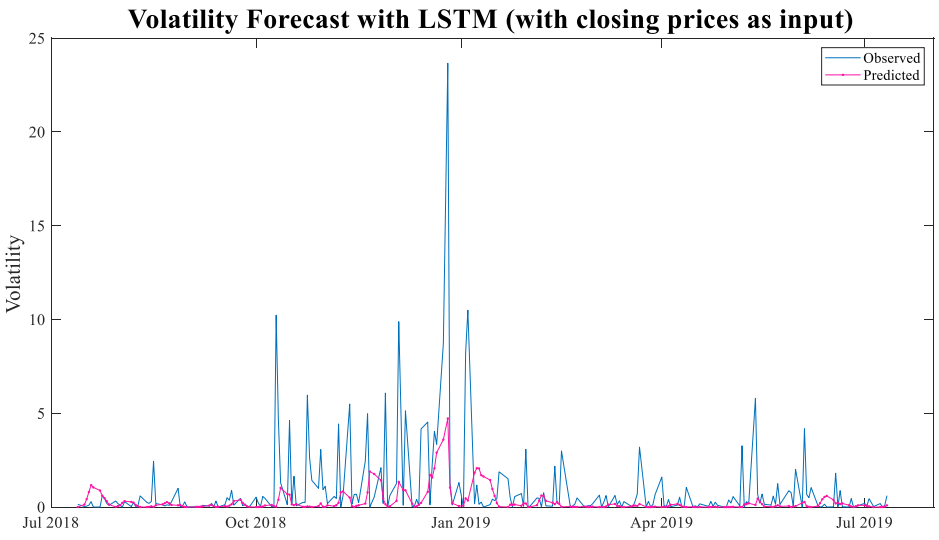


Figure 3.16: Volatility Forecast LSTM with CP as input (MATLAB)

Conclusion

This thesis aimed to demonstrate the superiority of the neural network in financial volatility forecasting for a univariate time series. The results achieved in the empirical model neither sustain nor contrasts this statement. The model proposed by Engle is very valid in measuring the oscillation in the price movements because it can capture almost all the key features of financial prices. However, under some circumstances, the LSTM outperformed the GARCH(1,1) in terms of mean squared error and root mean squared error. In experiment 3, for example, the neural network can predict the unseen part better than the heteroskedastic model. Besides, an interesting finding is that the LSTM network can achieve very good results even if the inputs are the closing prices. The simple GARCH model is lacking this ability since it requires a log transformation of the raw data to obtain satisfactory results in the out-of-sample analysis. The choice on which model is better to use depends on the type of input data and the task required. Certainly, the good outcomes achieved from the LSTM must be taken into consideration, even if, sometimes, they are a little worst than the classical model. In fact, it is admirable the power to achieve such values without requiring a priori assumption on the distribution of the errors and a priori knowledge on the form of the underlying function. Given its ability to understand any complex and non-linear connection behind the data, the long short-term memory network may offer higher performance in a multivariate analysis, where a huge dataset is required. In addition, one possible way to improve the predictive abilities of neural networks and offer better estimates may be the selection of the realized volatility, seen as mean of intraday observations of volatility, as a proxy of volatility.

Concluding, this thesis proves that the LSTM network can obtain meaningful information from noisy financial data. Possible future works may apply to stock index volatility prediction other recurrent neural network structures, such as Echo State Network, Neural Turing Machines and Continuous-time RNN, with the aim to enlarge this field of research and find the "perfect" model to forecast financial volatility.

Bibliography

Alpaydin, E. (2010). *Introduction to Machine Learning.* London: The MIT Press.

Arneric, J., Poklepovic, T., & Aljinovic, Z. (2014). GARCH based artificial neural networks in forecasting conditional variance of stoks returns. *Croatian Operational Research Review* (5), 329-343.

Balkin, S. D. (1997). *Using Recurrent Neural Networks for Time Series Forecasting.* Pennsylvania: University Park.

Bengio, Y., & Goodfellow, I. (2015). *Deep Learning.* Mit Press.

Bianchi, F. M., Maiorino, E., Kampffmeyer, M. C., Rizzi, A., & Jenssen, R. (2017). *Recurrent Neural Networks for Shoert-Term Load Forecasting.* Switzerland: Springer.

Bollerslev, T. (1986). Generalized Autoregressive Consitional Heteroskedasticity. *Journal of Econometrics, 31,* 307-327.

Bollerslev, T. (2007). *Glossary to ARCH (GARCH).* Duke University and NBER.

Brailsford, T., & Faff, R. (1992). An evaluation of volatility forecasting. *Journal of Banking and Finance, 20*(3), 419-438.

Bryden, J. (2014). *Biologically Inspired Computing: The Neural Network.* Leeds, England: University of Leeds - School of Computing.

Carroll, R., & Kearney, C. (2009). GARCH Modeling of Stock Market Volatility. In G. N. Gregoriou, *Stock Market Volatility* (pp. 71-91). New York: Chapman & Hall.

Danielsson, J. (2011). *Financial Risk Forecasting.* United Kingdom: Wiley.

Donaldson, G., & Kamstra, M. (1997). An artificial neural network-GARCH for international stock return volatility. *Journal of Empirical Finance, 4*(1), 17-46.

Engle, R. F., & Patton, A. J. (2001). *What Good Is A Volatility Model?* USA.

Fadda, S., & Can, M. (2017). Forecasting Conditional Variance of S&P100 Returns Using Feedforward and Recurrent Neural Networks. *Southeast Europe Journal of Soft Computing, 6*(1), 58-69.

Fischer, T., & Krauss, C. (2017). *Deep learning with long short-term memory networks for financial market predictions.* FAU Discussion Papers in Economics.

Francq, C., & Zakoian, J.-M. (2010). *GARCH Models. Structure, Statistical Inference and Financial Applications.* UK: John Wiley & Sons Ltd.

Gallo, C. (2005). *Artificial Neural Network in Finance Modelling.* Italy: Università di Foggia.

Gallo, C. (2012). *Costruzione di una Rete Neurale Artificiale per applicazioni Economico-Finanziarie.* Università degli Studi di Foggia.

Gencay, R., & Liu, T. (1996). *Nonlinear Modelling and Prediction with Feedforward and Recurrent Networks.*

Gurney, K. (1997). *An introduction to neural networks.* London: UCL Press.

Hansson, M. (2017). *On stock return prediction with LSTM networks.* Lund: Lund University.

Haykin, S. (2009). *Neural Networks and Learning Machines.* New Jersey: Pearson Education.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*(8), 1735-1780.

Kaastra, I., & Boyd, M. (1996). Designing a neural network for forecasting financial and economic time series. *Neurocomputing, 10*(3), 215-236.

Kotsiantis, S. B. (2007). Supervised Machine Learning: A Review of Classification. *Informatica*(31), 249-268.

Ladokhin, S. (2009). *Volatility Modeling in Financial Markets.* Amsterdam.

Lamoureux, C. G., & Lastrapes, W. D. (1990). Persistence in Variance, Structural Change and the GARCH Model. *Journal of Business & Economic Statistics, 8*(2), 225-234.

McNelis, P. D. (2005). *Neural Network in Finance: Gaining Predictive Edge in the Market.* USA: Elsevier.

Miah, M., & Rahman, A. (2016). Modelling Volatility of Daily Stock Returns: Is GARCH(1,1) Enough? *American Scientific Research Journal for Engineering, Technology, and Sciences, 18*(1), 29-39.

Monfared, S. A., & Enke, D. (2014). Volatility Forecasting using a Hybrid GJR-GARCH Neural Network Model. *Procedia Computer Science* (36), 246-253.

Namin Siami, S., & Namin Siami, A. (2018). *Forecasting Economic and Financial Time Series: ARIMA vs LSTM.* Texas: Texas Tech University.

Nelson, D. (1991). Conditional Heteroskedasticity in Asset Returns: A New Approach. *Econometrica, 59*(2), 347-370.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Determination Press. Retrieved from http://neuralnetworksanddeeplearning.com/index.html

Nwankpa, C. N., Ijomah, W., Gachagan, A., & Marshall, S. (2018). *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning.* Glasgow, UK: University of Strathclyde.

Olah, C. (2015). *Understanding LSTM Networks.* Retrieved from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Ormoneit, D., & Neuneier, R. (1996). Experiments in predicting the German stock index DAX with density estimating neural networks. *Conference on Computational Intelligence for Financial Engineering* . IEEE/IAFE .

Patton, A. (2010). Volatility forecast comparison using imperfect volatility proxies. *Journal of Econometrics*.

Poon, S., & Granger, C. (2003). Forecasting Volatility in Financial Markets: A Review. *Journal of Economic Literature, XLI*, 478-539.

Poon, S., & Granger, C. (2005). Practical Issues in Forecasting Volatility. *Financial Analysts Journal, 61*(1), 45-56.

Poon, S.-H. (2005). *A Practical Guide to Forecasting Financial Market Volatility.* England: John Wiley & Sons Ltd.

Ruder, S. (2016). *An overview of gradient descent optimization algorithms.* Dublin: Insight Centre for Data Analytics, .

Sharma, S. (2017, Sep 6). *Activation Functions in Neural Networks*. Retrieved from Towards Data Science:

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Taylor, S. (2004). *Asset Price Dynamics, Volatility, and Prediction.* USA: Princeton University Press.

Taylor, S. J. (2008). *Modelling Financial Time Series.* World Scientific Publishing Co.

The MathWorks. (2019). *Time Series Forecasting Using Deep Learning*. Retrieved 07 2019, from MathWorks: https://it.mathworks.com/help/deeplearning/examples/time-series-forecasting-using-deep-learning.html;jsessionid=7c552caf113e278c66a53ef2a09a

Vitali, G. (xszj). bfbhdf. *jdcbhbcjsc*.

Wennstrom, A. (2014). *Volatility Forecasting Performance: Evaluation of GARCH type volatility models on Nordic equity indices.* Stockholm: Royal Institute of Technology.

Yu, H., & Wilamowski, B. M. (2010). Levenberg–Marquardt Training. In *Intelligent System.*

Zhang, G. (1998). Forecasting with artificial neural networks: the state of art. *International Journal of Forecasting*(14), 35-62.