LIBERA UNIVERSITÀ INTERNAZIONALE DEGLI STUDI SOCIALI
"LUISS - GUIDO CARLI"

Department of Economics and Finance
Chair of Asset Pricing

MACHINE LEARNING FOR VOLATILITY FORECASTING

Supervisor:
Prof. Emilio Barone

Co-Supervisor:
Prof. Nicola Borri

Candidate:
Simone Romano

Academic Year 2019-2020

Contents

Introduction

Volatility is a central topic in the financial literature and such paramount importance lies in the vast array of its applications.

For example, volatility represents an essential element to many investment decisions as it is often taken as the starting point for optimal portfolio allocations. In 1952, Harry Markowitz laid the foundation of the Modern Portfolio Theory by which investors are risk averse and have a utility function increasing with expected return and decreasing with volatility.

Volatility is also a key input in the pricing of many derivatives. Indeed, to evaluate options' fair value, the volatility of the underlying asset until the expiration date must be known. Besides, in recent years, even derivatives with volatility itself as the underlying have been introduced, and in these cases, the definition and measurement of volatility must be specified in the derivative contracts.

In risk management, volatility is relevant for computing the Value at Risk (VaR), whose estimation has become a standard practice for financial institutions. Indeed, since the first Basel Accord was established in 1996, banks and trading venues are required to set aside a reserve capital of at least three times that of VaR.

Volatility can also have wide consequences on the economy as a whole. Thus, policymakers regard volatility as an indicator of uncertainty in the financial market. For example, both the Federal Reserve and the Bank of England take into account the securities volatility in establishing their monetary policies. Besides, volatility negatively affects market liquidity since when the former spikes, the latter usually declines.

Volatility is crucial for hedging strategies as well. Indeed, during stressed market conditions, not only volatility increases but also correlations among different secu-

rities do so. In these circumstances, derivative instruments may work as insurance against sudden market downturns.

For all these reasons, the relevance of volatility forecasting follows as a natural consequence. Although the literature on this subject is extensive and many models have been proposed, in the last years, we have been assisting to a rise in the applications of machine learning techniques to many different sectors. Hence, this thesis is aimed at bridging the gap between the classical financial literature and the machine learning models for volatility forecasting.

The rest of the work is organized as follows: in the first chapter, we will provide the reader with a background of volatility together with some stylized facts from the S&P 500 Index.

In the second chapter, we will describe the most relevant econometric models for volatility forecasting. The Exponential Weighted Moving Average (EWMA) model, the AutoRegressive Conditional Heteroscedasticity (ARCH) model, and the Generalized AutoRegressive Conditional Heteroscedasticity (GARCH) model will be presented. Afterwards, the model parameters estimation through the maximum likelihood method will be illustrated.

In the third chapter, we will introduce some machine learning models. In particular, being the best suited for regression analyses, we will circumscribe our discussion to Artificial Neural Networks (ANNs). We will describe them in comparison, and as alternative tools, to econometric models. After a general description of ANNs, we will see in particular the Feedforward Neural Network (FNN), the Recurrent Neural Network (RNN), and the Long Short Term Memory (LSTM).

Finally, in the last chapter, we will describe the steps taken to build a machine learning model and compare its predictive power to that of the presented econometric models.

Chapter 1

Volatility

1.1 Defining and measuring volatility

The volatility of a security or a market index is a measure of the dispersion of its returns across their mean. Usually denoted by σ, it is defined as the standard deviation of logarithmic returns observed over fixed time intervals:

$$\sigma = \sqrt{\frac{1}{n}\sum_{t=1}^{n}(r_t - \mu)^2}$$

where:

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

S_t is the price at time t

μ is the mean of r_t

n is the number of observations

Estimating the population variance from the sample variance can be done with the following unbiased estimator:

$$s^2 = \frac{n}{n-1}\sigma^2$$

Although it is common practice to take the square root to get the volatility, it should be noted that this gives an estimate which is biased low. Indeed, since the square root is a strictly concave function, by Jensen's inequality:

$$\mathbb{E}\left[\sqrt{s^2}\right] \leq \sqrt{\mathbb{E}\left[s^2\right]} = \sqrt{\sigma^2} = \sigma$$

Assuming returns are normally distributed, Holtzman (1950) proved that:

$$\mathbb{E}[s] = c(n)\sigma$$

where

$$c(n) = \sqrt{\frac{2}{n-1}}\frac{\Gamma(\frac{n}{2})}{\Gamma(\frac{n-1}{2})}$$

where $\Gamma(\cdot)$ is the gamma function.

Figure 1.1 shows the size of the correction factor $c(n)$ for different sample sizes.

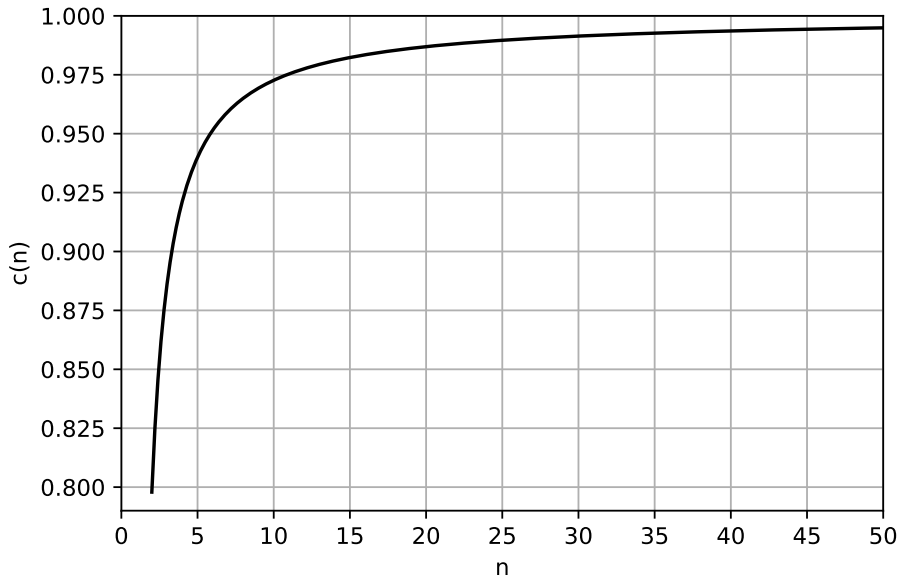

Figure 1.1: Correction factor to obtain an unbiased estimate of σ

Then, an unbiased estimator, $\tilde{s}$, of the population standard deviation could be obtained by dividing s by $c(n)$.

$$\tilde{s} = \frac{s}{c(n)}$$

However, when comparing s with $\tilde{s}$ by their Mean Squared Error (MSE)[1], the former is nevertheless to be preferred (see Figure 1.2). Indeed, despite $\tilde{s}$ produces estimates that are on average correct, its higher variance makes it converge to the true volatility more slowly than s.

[1] $\text{MSE}(\hat{\theta}) = \mathbb{E}\left[(\hat{\theta}-\theta)^2\right] = \text{Var}(\hat{\theta}-\theta) + \mathbb{E}\left[\hat{\theta}-\theta\right]^2 = \text{Var}(\hat{\theta}) + \text{bias}^2$

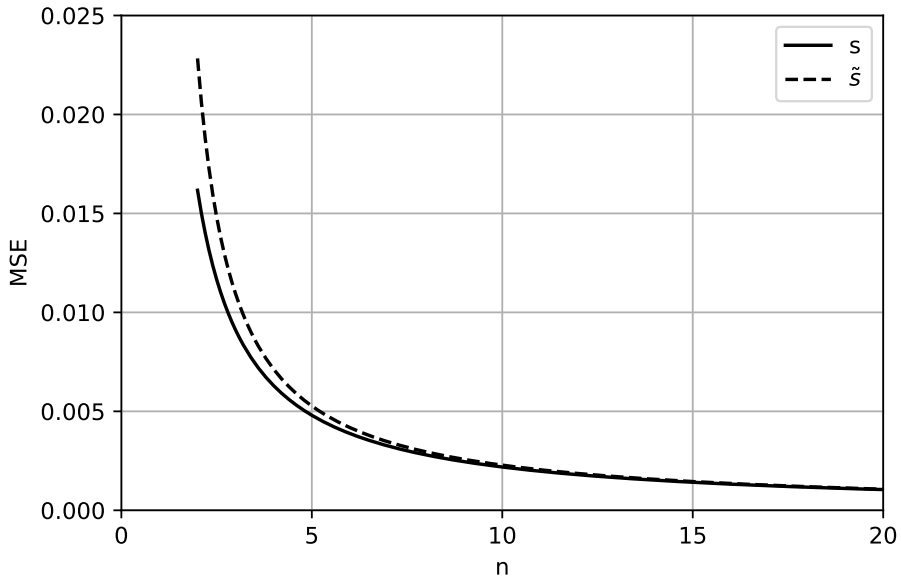

Figure 1.2: MSE for two volatility estimators. A value of 20% has been assigned to σ for illustrative purposes.

Both the estimators suggest that using more data would increase the accuracy of the estimation. That would not be a problem if we were measuring the volatility of an unchanging process. Though, volatility is not constant over time and using too much data could lead at including information no longer relevant to the current state of the market. A good compromise could be found by using the last 30-60 closing prices from daily data.

By convention, volatility is expressed in annual terms. Since most of the variation in securities' returns come from trading activity, practitioners consider a year as formed by 252 days, which is usually the number of days stocks are traded within a year. Assuming independent returns and constant volatility, the annual volatility can then be calculated as:

$$\sigma_{annual} = \frac{\sigma_\tau}{\sqrt{\tau}}$$

where τ is the length of time interval in years. For example, if we had to express daily volatility per annum:

$$\sigma_{annual} = \sigma_{daily} \cdot \sqrt{252}$$

The aforementioned volatility is usually referred to as historical volatility. There exist other two types of volatility: the realized volatility and the implied volatility.

1.2 Realized volatility

When computing the historical volatility, the mean return needs to be estimated. Although its sample estimator is unbiased, estimates of the mean return are noisy, especially for small samples. Besides, even increasing the sample size, Merton (1980) noted that periods when a security experiences a downtrend result in a negative average return which conflicts with the prior non-negativity restriction to the ex-ante expected return. In light of these shortcomings, many authors and professionals set the mean return to zero. This type of volatility is referred to as *realized volatility*.

However, with the availability of high frequency data, the term is more often associated with the estimation of volatility using intraday returns proposed by Andersen and Bollerslev (1998). The intuition behind the use of realized volatility is conveyed within the popular continuous-time diffusion setting:

$$ds(t) = \mu(t)dt + \sigma(t)dW(t) \qquad t \geq 0$$

where $s(t)$ is the logarithmic asset price at time t, $\mu(t)$ is the drift term and $\sigma(t)$ is the instantaneous volatility that inflates the change in price relative to $dW(t)$, which is a stochastic Brownian step. The continuously compounded return over the time interval from $t - \delta$ to t, $0 \leq \delta \leq t$, is therefore:

$$r(t, \delta) = s(t) - s(t - \delta) = \int_{t-\delta}^{t} \mu(\tau)d\tau + \int_{t-\delta}^{t} \sigma(\tau)dW(\tau)$$

and its quadratic variation $QV(t, \delta)$:

$$QV(t, \delta) = \int_{t-\delta}^{t} \sigma^2(\tau)d\tau$$

Considering a discrete partition $\{t - \delta + \frac{j}{n}, j = 1, ...n \cdot \delta\}$ of the $[t - \delta, t]$ interval,

the realized volatility, RV, is defined as

$$RV(t,\delta;n) = \sqrt{\sum_{j=1}^{n\cdot\delta} r^2(t-\delta+\frac{j}{n},\frac{1}{n})}$$

If the returns are serially uncorrelated and the sample path for σ is continuous, it follows by the theory of quadratic variation (see Karatzas and Shreve (1988)), that

$$RV^2(t,\delta;n) \xrightarrow{p} QV(t,\delta), \qquad \text{as } n \to \infty$$

Hence, RV^2 approximates QV as the sampling frequency n increases. However, this result presents some issues. First, continuous price recording is unfeasible in practice even for the most liquid assets. This gives rise to measurement errors due to the discretization of the observations. Second, intraday returns can be affected by spurious autocorrelations due to many microstructure effects, like noise trades, price discreteness, bid-ask bounces, market fragmentation... etc. These autocorrelations may lead to an overestimation of the RV and give rise to a trade-off between bias and variance. Indeed, although the highest possible sampling frequency should be adopted to improve the efficiency, that would induce biased RV estimates. In order to deal with these issues, 5-15 minutes returns are then commonly used to compute realized volatility.

1.3 Implied volatility

Both the historical and the realized volatility are computed using past observations. Implied volatility (IV), instead, is derived from current option prices observed in the market and represents investors' expectations about the future volatility of the underlying asset. For that reason, in contrast with the other two types of volatility, IV is also said to be a forward looking measure.

Once specified an option pricing model, the implied volatility is the value that equates the theoretical to the observed option price. The most used model to price European options written on stocks paying no dividends is the Black–Scholes–Merton

model, according to which the price of a call and a put, denoted by c and p respectively, is equal to

$$c = S_0 N(d_1) - K e^{-rT} N(d_2) \tag{1.1}$$

$$p = K e^{-rT} N(-d_2) - S_0 N(-d_1) \tag{1.2}$$

where

$$d_1 = \frac{ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}$$

$$d_2 = \frac{ln(S_0/K) + (r - \sigma^2/2)T}{\sigma\sqrt{T}} = d_1 - \sigma\sqrt{T}$$

where S_0 is the stock price at time zero, K is the strike price, r is the continuously compounded risk-free rate, σ is the stock price volatility, T is the time to maturity of the option and $N(\cdot)$ is the cumulative distribution function of the standard normal distribution.

Unfortunately, it is not possible to invert Equations (1.1) and (1.2) so that σ is expressed as a function of the other parameters. However, being the options prices monotonic in volatility, the IV can be easily found by numerical search methods.[2]

According to the Black-Scholes-Merton model, all European options written on the same underlying and having the same time to maturity should have the same implied volatility. Actually, that is not true in practice. When implied volatility is plotted against moneyness (K/S), the resulting graph is typically convex and the phenomenon is so systematic that it is referred to as *volatility smile*. The explanation of such an inconsistency lies in the fact that the implied (and actual) distribution of returns is not correctly captured by the model (see Section 1.5) and that the future volatility is uncertain. Besides, errors in the implied volatility estimation may arise from asynchronous reporting of options prices and underlying asset price. Thus, the existence of multiple implied volatilities, one for each strike price, has led to the need for a single measure, and so to the introduction of the VIX Index.

[2] One efficient way to find the implied volatility is to use Newton-Raphson method. It consists in setting an initial estimate σ_0 of the volatility and producing better estimates $\sigma_1, \sigma_2, ...$ using the formula $\sigma_{i+1} = \sigma_0 - f(\sigma_i)/f'(\sigma_i)$ where $f(\sigma_i)$ is the difference between the theoretical price of the option when $\sigma = \sigma_i$ and its market price, $f'(\sigma_i)$ is the first derivative of the option value with respect to σ_i, also known as *vega*.

1.4 The VIX Index

The VIX Index, also referred to as the *fear index* or *fear gauge*, is an index of the implied volatility of 30-day options on the S&P 500 Index (ticker symbol SPX). It was introduced in 1993 by the Chicago Board Options Exchange (CBOE). In its original formulation developed by Whaley (1993), the index was based on the option prices of eight at-the-money calls and puts written on the S&P 100 (ticker symbol OEX). In 2003, since most of the trading volume shifted to SPX and out-of-the-money put options also became largely bought for insurance purposes, the CBOE updated the VIX calculation to account for these changes. Since 2004, VIX is also tradable through futures contracts and since 2006 through options contracts as well.

Unlike to the usual calculation of implied volatility, the VIX is not computed using any option pricing model but it derives IV directly from option prices belonging to one of two consecutive maturities between 23 and 37 days. The variances, σ_1 and σ_2, resulting from near-term and next-term options respectively, are then interpolated in order to return the 30-day implied volatility. The current formula to compute the VIX is the following:

$$VIX = 100\sqrt{\left[T_1\sigma_1^2\left(\frac{N_{T_2}-N_{30}}{N_{T_2}-N_{T_1}}\right)+T_2\sigma_2^2\left(\frac{N_{30}-N_{T_1}}{N_{T_2}-N_{T_1}}\right)\right]\frac{N_{365}}{N_{30}}}$$

where:

$$\sigma_j^2 = \frac{2}{T_j}\sum_i \frac{\Delta K_i}{K_i^2}e^{r_j T_j}Q(K_i) - \frac{1}{T_j}\left(\frac{F_j}{K_0}-1\right)^2, \qquad j \in \{1,2\}$$

$j = 1$ and $j = 2$ indicate near-term and next-term options respectively

T_j is the time to maturity for j-type options

F_j is the forward index level derived from j-type index option prices

K_0 is the first strike below the forward index level, F_j

K_i is the strike price of i^{th} option having a non-zero bid price

$\Delta K_i = (K_{i+1} - K_{i-1})/2$

r_j is the risk-free interest rate to expiration for j-type options

$Q(K_i)$ is the midpoint of the bid-ask spread for each option with strike K_i
N_{T_j} is the number of minutes to settlement of j-type options
N_{30} and N_{365} are the number of minutes in 30 and 365 days respectively

Figure 1.3 shows the VIX Index between January 1990 and December 2020. At the beginning of the '90s, the Asian financial crisis and the collapse of Long Term Capital Management (LTCM) can be seen from the highs. In the early 2000s, the burst of the dot-com bubble caused the index to peak again. After readjusting to normal levels in the following years, the VIX touched one of its major peaks in 2008, during the global financial crisis. Then, the flash crash, the Euro area crisis and the oil crisis led to further spikes in the implied volatility in 2010, 2012 and 2016 respectively. Finally, the last spike refers to the COVID-19 crash in March 2020.

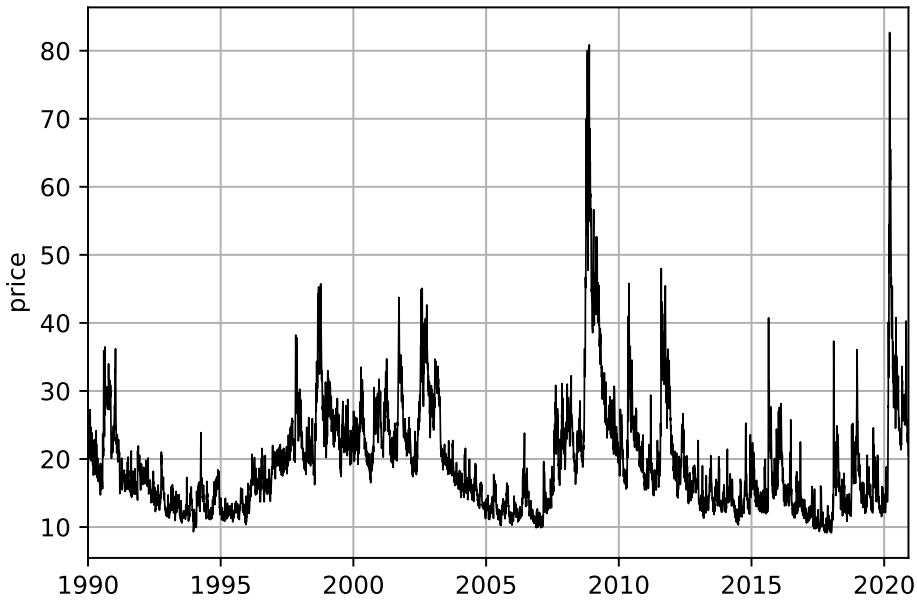

Figure 1.3: The VIX index, January 1990 to December 2020

1.5 Stylized facts: evidence from the S&P 500

Many stylized facts about volatility have emerged over the years and been confirmed in numerous studies. A good volatility model, then, must be able to capture and reflect these stylized facts. Without loss of generality, we will show some evidence from the S&P500, sometimes looking at volatility itself, sometimes analyzing the returns, other times including both in our study. All data are downloaded from CBOE and refer to the period from January 1990 to December 2020.

Starting from returns, many models assume they are normally distributed. However, when compared with the normal density function, actual returns distribution has fatter tails and slight negative skewness meaning that large moves occur more frequently than expected according to the normal distribution, and that when these occur, they are more likely to be negative (see Figure 1.4).

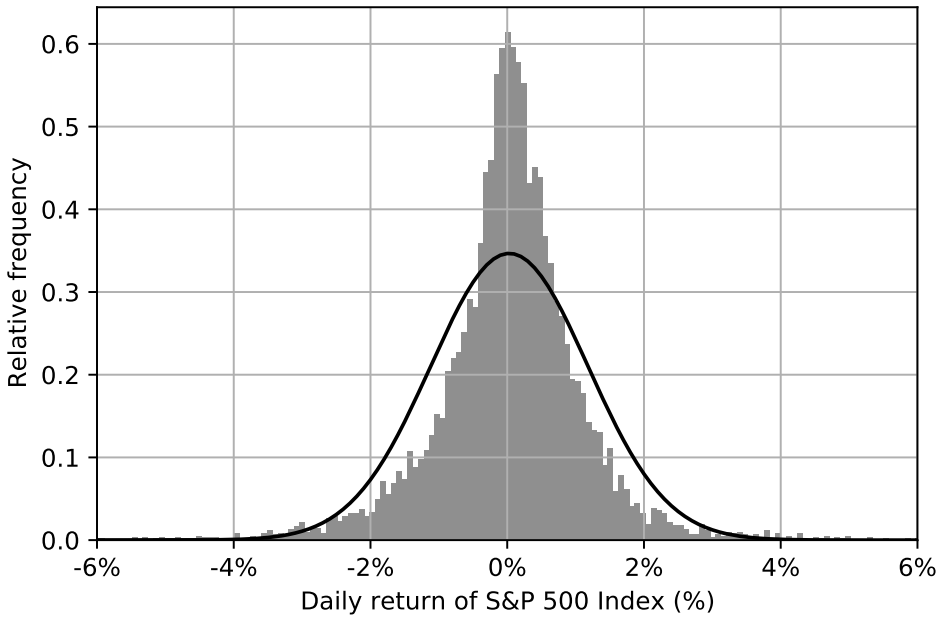

Figure 1.4: Distribution of daily S&P 500 returns

The presence of fat tails is more evident in Figure 1.5, where the quantiles of the standardized actual and normal distributions are compared.

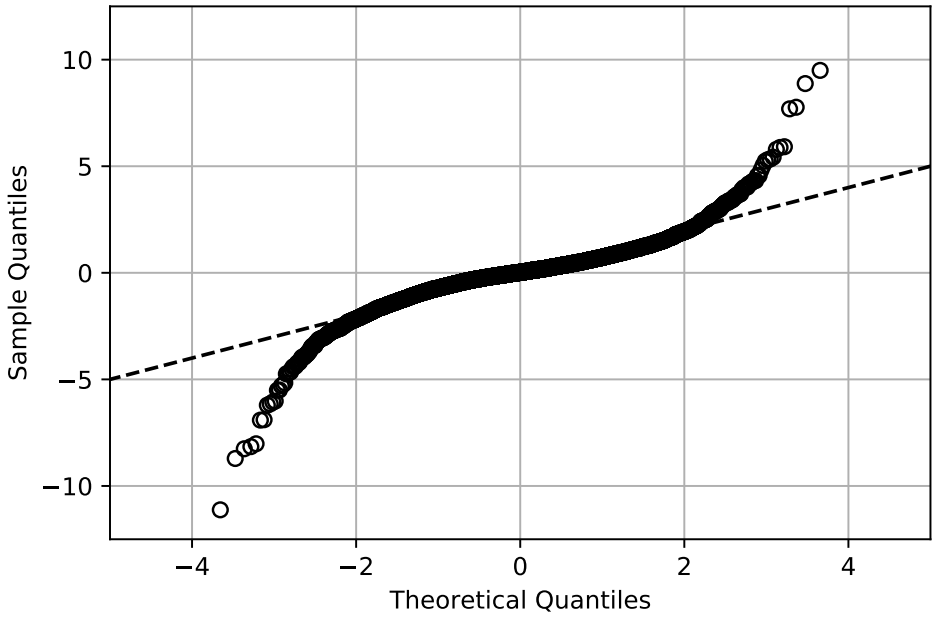

Figure 1.5: Q-Q plot for standardized daily S&P500 log returns

Returns are notoriously difficult to predict by simply looking at their past values. The same cannot be said about absolute returns, though. An analysis of the correlations between weekly returns across multiple lags is presented in Figure 1.6. It shows that returns are persistent in their absolute value and that this serial dependency decays slowly over time.

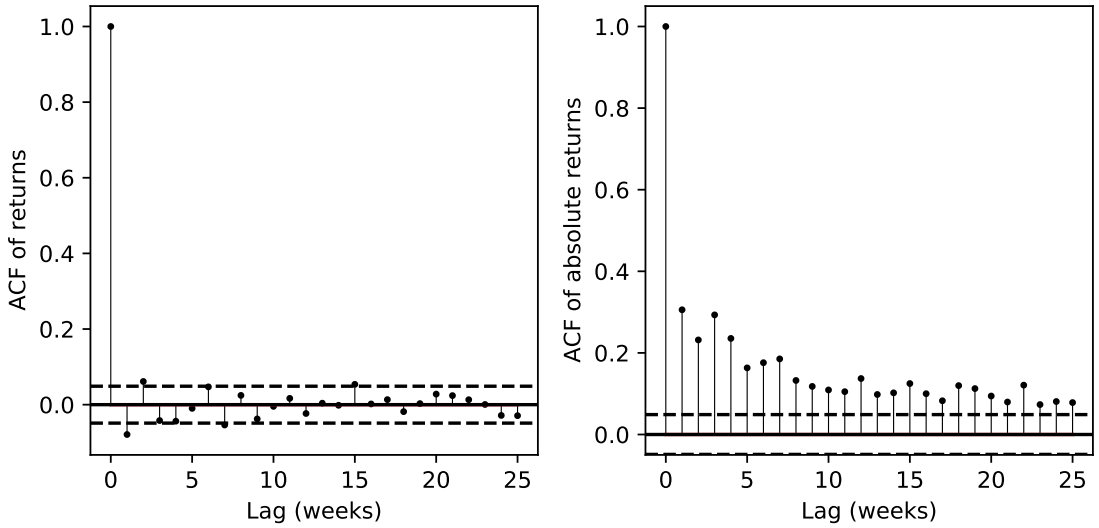

Figure 1.6: Autocorrelation function (ACF) for weekly S&P 500 returns (left) and absolute returns (right)

In other terms, as described by Mandelbrot (1963), who first noted such returns

behaviour, "large changes tend to be followed by large changes - of either sign - and small changes tend to be followed by small changes".

The phenomenon, concerning the amplitude of returns, reflects also in volatility and is known as *volatility clustering*. Then, volatility exhibits a long memory as well, and such a feature reveals particularly important when making forecasts. Figure 1.7 illustrates how the current volatility is a good estimate of next month's volatility.

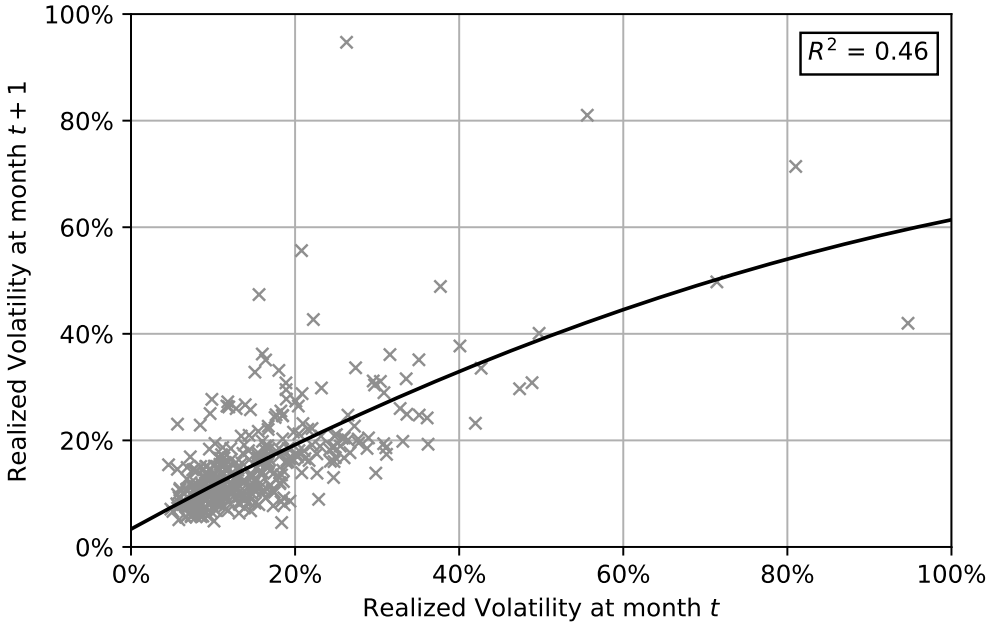

Figure 1.7: Clustering effect in realized volatility

However, the relationship is not linear: the higher the current volatility, the higher the expected divergence next month from its actual value. This is due to another stylized fact about volatility, i.e. its mean reverting behaviour. In order to give a better look to such a property, we have taken two buckets from realized volatilities representing the highest and lowest 10% respectively of all the historical observations and we monitored how the average volatility of each bucket has changed through time after initially observed within the top/bottom 10% of the distribution (see Figure 1.8).

Besides the slow convergence of both the buckets to the long-term average volatility, a different pattern can be detected: in line with Figure 1.7, observations in the top bucket tend faster to the mean volatility. This suggests that low levels of volatility are much more stable and persistent than high ones.

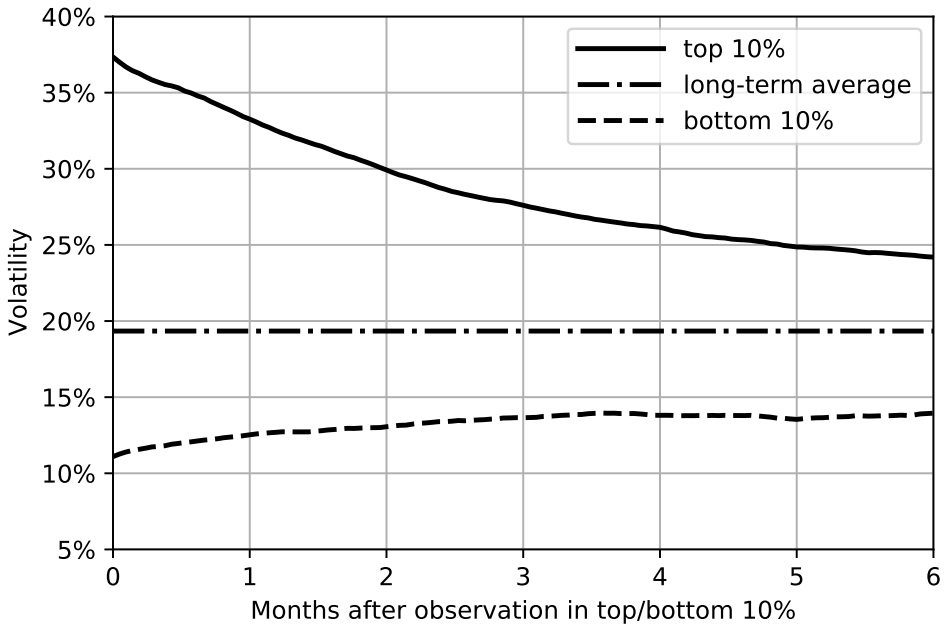

Figure 1.8: Volatility converges towards long-term average

Another feature of volatility can be inferred by Figure 1.8 and regards its distribution. Indeed, volatility is heavily skewed to the right with many more periods of high volatility that we would expect if the distribution was normal. This is better shown in Figure 1.9.

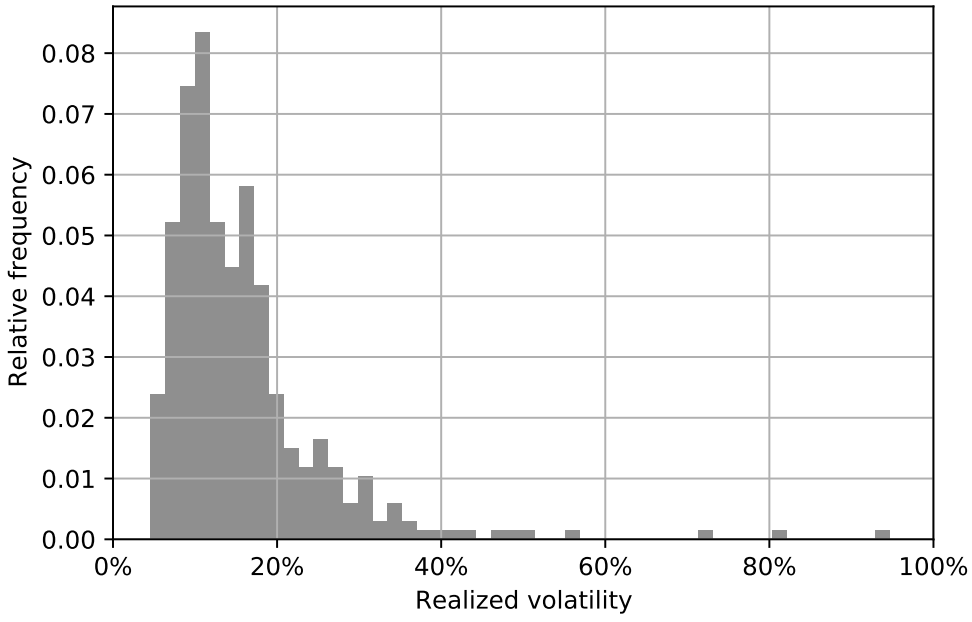

Figure 1.9: Distribution of realized volatility

However, in line with the evidence found by Andersen et al. (2001) using intraday returns, the distribution of logarithmic monthly realized volatility approximates the normal distribution (see Figure 1.10).

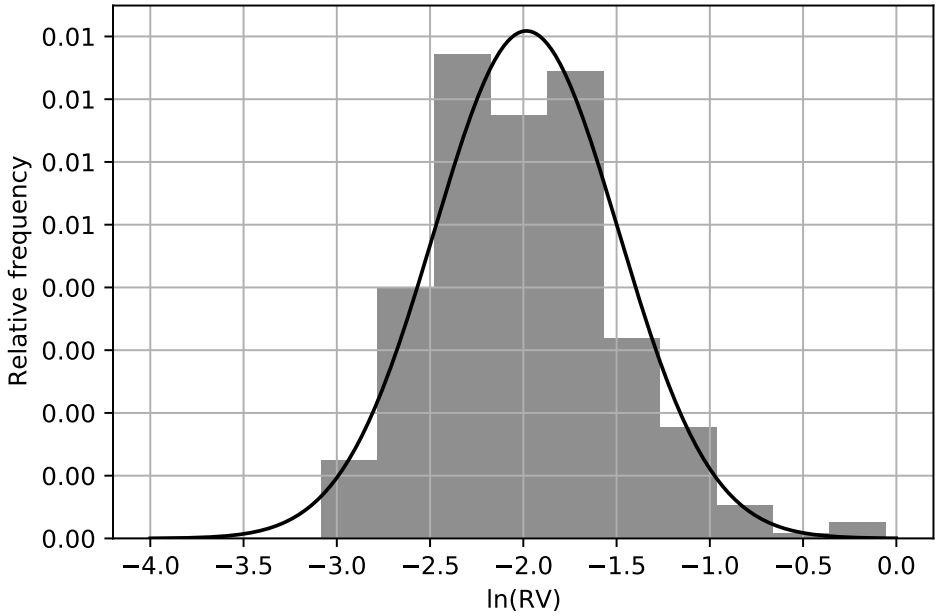

Figure 1.10: Distribution of logarithmic realized volatility

Another well-known stylized fact about volatility regards its negative relationship with equity returns. Also in this case the relationship is not linear: the greater the percentage fall in prices, the higher the change in volatility. Although the phenomenon was first noted by Black (1976) in realized volatility, it concerns implied volatility as well. In order to avoid measurement errors from the realized volatility estimation, Figure 1.11 compares the daily returns of the S&P 500 with the daily percentage change in VIX.

The reason for the negative relationship has been the subject of much research. The explanation suggested by Black was that when stock prices decline, companies become more leveraged and perceived riskier, therefore more volatile. Hence the term *leverage effect* often used to refer to that phenomenon. An alternative explanation, known under the name of *volatility feedback effect*, reverts the causality: when volatility increases, higher rates of returns are demanded causing stock prices to decline. Both the hypothesis have been explored by a number of authors and, on balance, the empirical evidence appears to favour the volatility feedback effect.

Indeed, the negative relationship holds also in companies which have low leverage.

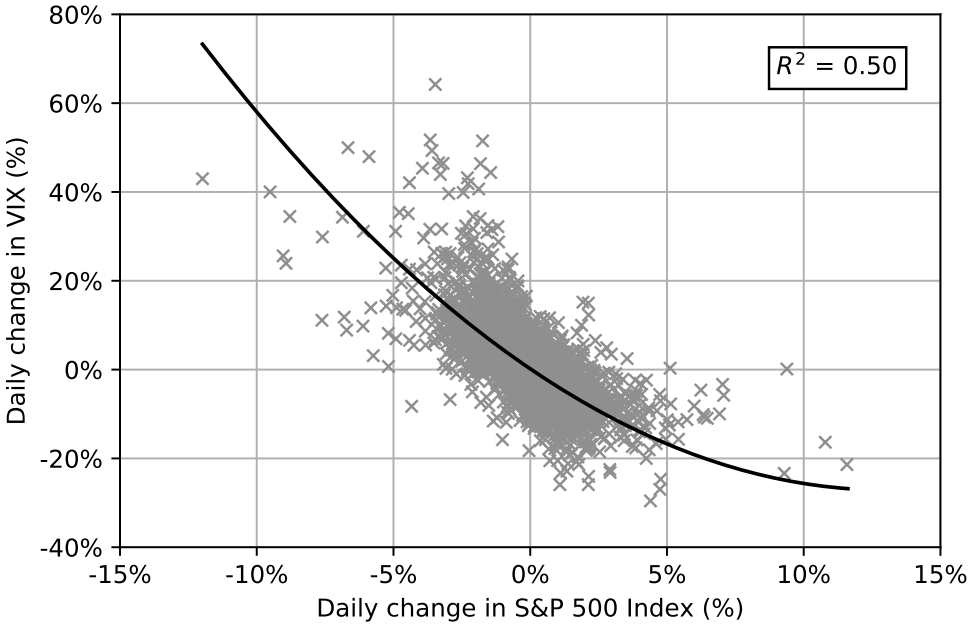

Figure 1.11: Relationship between S&P 500 daily returns and daily percentage changes in VIX

Finally, comparing through time the implied volatility (measured by the VIX), to the realized volatility, the former appears systematically higher than the latter (see Figure 1.12).

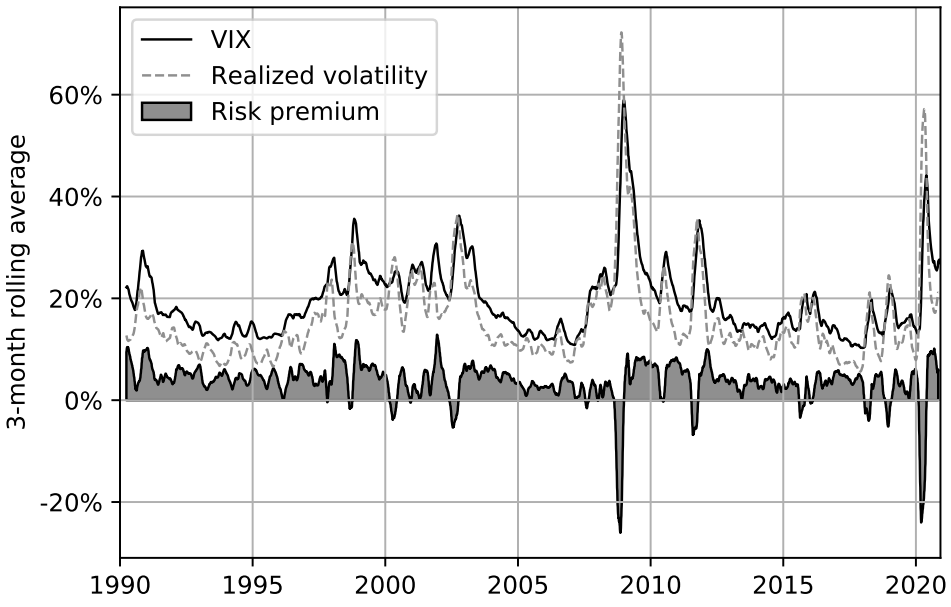

Figure 1.12: Volatility risk premium

Such a positive difference between the two variables has then been explained as a risk premium. Since options can provide protection against losses in periods of market downturns, sellers demand a higher price to be compensated for the risk of large movements in the underlying asset. In other terms, the volatility risk premium can be regarded as an insurance premium demanded by options underwriters to investors willing to hedge their portfolios. However, the premium has not always been positive. Indeed, during the global financial crisis and the COVID-19 crash, realized volatility spiked and overcame past investors' expectations about volatility.

Chapter 2

Econometric Models

2.1 The EWMA model

In the last chapter, we saw that absolute returns display high and significant autocorrelations which cause volatility to cluster. The effect on volatility may be more evident from Figure 2.1, which shows that squared returns exhibit slowly decaying serial dependency as well.

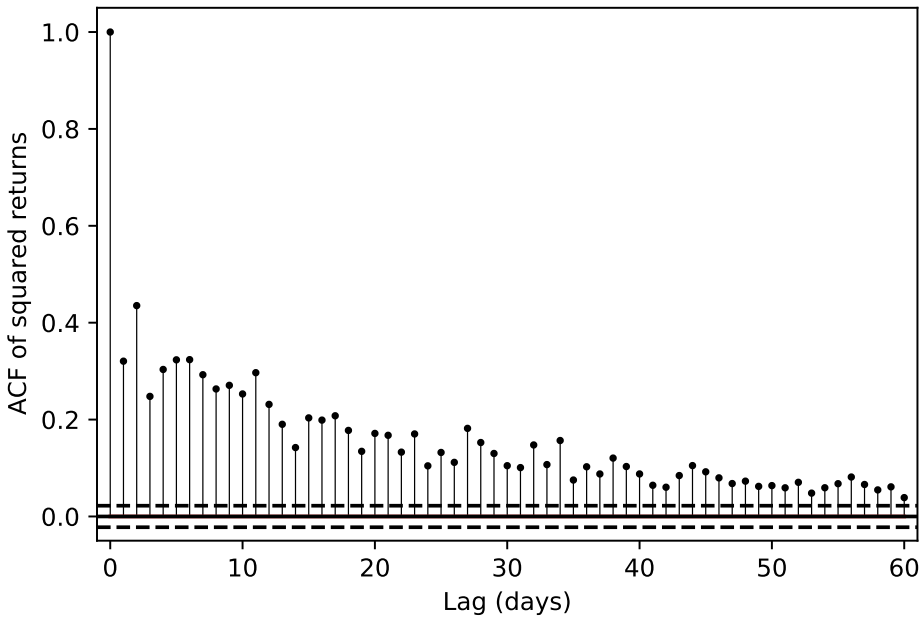

Figure 2.1: Autocorrelation function (ACF) for daily S&P 500 squared returns

The Exponential Weighted Moving Average (EWMA) model takes into account this behaviour. Its application to volatility modelling and forecasting was first in-

troduced by JP Morgan's RiskMetrics system.
According to the model, assuming

$$\epsilon_t = r_t - \mu_t = \sigma_t z_t, \qquad z_t \sim \text{IID } \mathcal{N}(0,1) \tag{2.1}$$

the conditional variance is expressed as:

$$\sigma_t^2 = (1-\lambda)\sum_{i=1}^{\infty} \lambda^{i-1}\epsilon_{t-i}^2, \qquad \lambda \in (0,1) \tag{2.2}$$

where ϵ_t is the demeaned return at time t, z_t is a Gaussian white noise with unit variance and λ is the decay factor. In practice, for the same reasons mentioned in Section 1.2, μ is set to zero and so:

$$r_t = \epsilon_t \tag{2.3}$$

Then, the model forecasts variance from averaging squared returns in the past and it implies that recent returns are more informative than distant ones. Indeed, it attributes weights to past squared returns that exponentially decrease as we move backwards in time. They depend just on λ and the lower its value, the higher the importance the model gives to recent observations (see Figure 2.2).

Equation (2.2) is not the common form in which the EWMA model is usually known and used, though. Indeed, rearranging σ_t^2 and σ_{t-1}^2 as follows,

$$\begin{aligned}
\sigma_t^2 &= (1-\lambda)r_{t-1}^2 + (1-\lambda)\sum_{i=2}^{\infty}\lambda^{i-1}r_{t-i}^2 \\
&= (1-\lambda)r_{t-1}^2 + (1-\lambda)\sum_{i=1}^{\infty}\lambda^{i}r_{t-1-i}^2 \\
&= (1-\lambda)r_{t-1}^2 + \lambda\frac{(1-\lambda)}{\lambda}\sum_{i=1}^{\infty}\lambda^{i}r_{t-1-i}^2
\end{aligned} \tag{2.4}$$

$$\sigma_{t-1}^2 = \frac{1}{\lambda}(1-\lambda)\sum_{i=1}^{\infty}\lambda^{i}r_{t-1-i}^2 \tag{2.5}$$

And substituting Equation (2.5) into Equation (2.4), we arrive at a much simpler

formula:

$$\sigma_t^2 = (1 - \lambda) r_{t-1}^2 + \lambda \sigma_{t-1}^2 \tag{2.6}$$

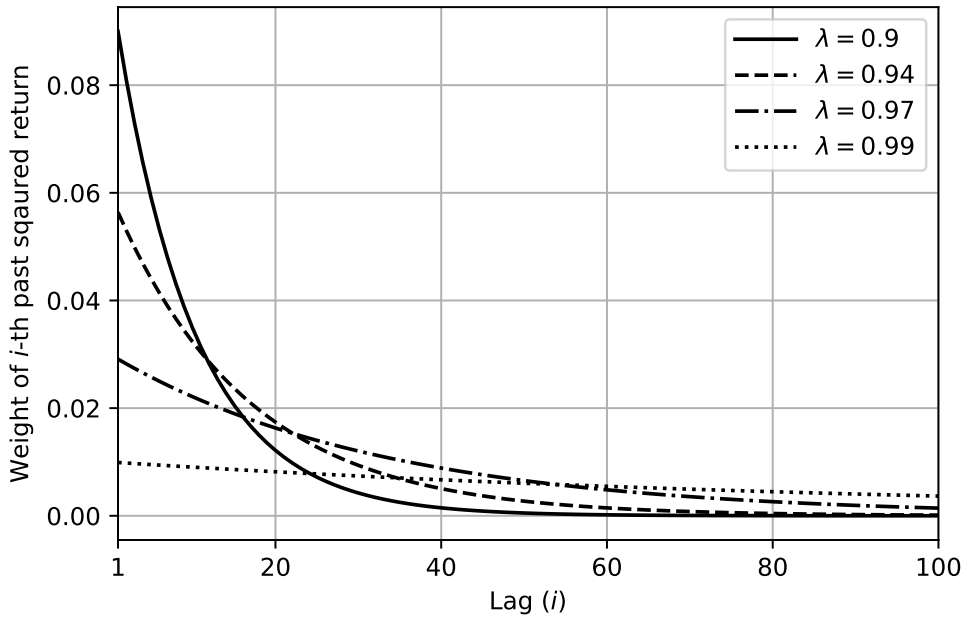

Figure 2.2: Weights of past observations for different values of λ

Equation (2.6) turns out appealing to practitioners, indeed, at any given time, only the current estimate of the variance and the most recent return need to be stored to forecast the volatility.

That is not the only attractive feature it has. Indeed, the model contains only one parameter to be estimated: λ. In 1996, RiskMetrics found that the estimates for λ were quite similar across 480 different assets and therefore suggested to set the decay factor to 0.94 and 0.97 for daily and monthly data respectively.

2.2 The ARCH model

The EWMA model captures the volatility clustering but it does not account for the tendency of volatility to return to its long-term average. The AutoRegressive Conditional Heteroscedasticity (ARCH) model, instead, was proposed by R. F. Engle (1982) to capture both the stylized facts. Assuming Equations (2.1) and (2.3), the

structure of the model in its simplest form, denoted by ARCH(1), is represented as follows:

$$\sigma_t^2 = \omega + \alpha r_{t-1}^2 \tag{2.7}$$

Although, on the one hand, the EWMA model in Equation (2.6) can be seen as a special case of ARCH(1) with $\omega = 0$, on the other hand, the ARCH model gives no weight to the current volatility estimate and relies just on the last squared return. Not surprisingly, this setting results in less accurate forecasts with respect to the EWMA model and so more past observations are usually included in the ARCH model. Indicating with q the number of included past returns, then the ARCH(q) model becomes:

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i r_{t-i}^2 \tag{2.8}$$

Sometimes, Equation (2.8) is also represented with $\omega = \gamma V_L$ in order to show that the model assign a weight (γ) to the long-run variance (V_L) as well:

$$\sigma_t^2 = \gamma V_L + \sum_{i=1}^{q} \alpha_i r_{t-i}^2 \tag{2.9}$$

However, for the purpose of parameters estimation, Equation (2.8) is usually used, and the long-run volatility can then be computed by finding the unconditional variance. Indeed, since $\mathbb{E}\left[r_t^2\right] = \mathbb{E}\left[\sigma_t^2 z_t^2\right] = \mathbb{E}\left[\sigma_t^2\right]\mathbb{E}\left[z_t^2\right] = \mathbb{E}\left[\sigma_t^2\right] \cdot 1 = \mathbb{E}\left[\sigma_t^2\right]$, setting $V_L = \mathbb{E}\left[\sigma_t^2\right]\ \forall t$,

$$\begin{aligned} V_L &= \omega + \sum_{i=1}^{q} \alpha_i \mathbb{E}\left[r_{t-i}^2\right] = \omega + \sum_{i=1}^{q} \alpha_i \mathbb{E}\left[\sigma_{t-i}^2\right] \\ &= \omega + V_L \sum_{i=1}^{q} \alpha_i = \frac{\omega}{1 - \sum_{i=1}^{q} \alpha_i} \end{aligned} \tag{2.10}$$

Because both the conditional and unconditional variances must be positive numbers, the following conditions must holds: $\omega \geq 0$ and $\sum_{i=1}^{q} \alpha_i < 1$.

2.3 The GARCH model

Although the ARCH(q) model could seem a simple way to forecast volatility, accounting both for the clustering and mean reverting behaviour of volatility, from a statistical perspective it is not as innocuous as it may seem. Indeed, even after individuating the correct amount of q observations to include within the model, as q increases, the parameters result hard to estimate and unstable in forecasting. The Generalized AutoRegressive Conditional Heteroscedasticity (GARCH) model was proposed by Bollerslev (1986) in order to deal with these shortcomings.

Its general formulation, denoted by GARCH(p,q), is represented as follow:

$$\sigma_t^2 = \omega + \sum_{i=1}^{q} \alpha_i r_{t-i}^2 + \sum_{i=1}^{p} \beta_j \sigma_{t-i}^2 \tag{2.11}$$

where $\omega, \alpha_i, \beta_j > 0 \, \forall i, j$ and the same assumptions made for ARCH and EWMA must hold.

Analogously to the ARCH model, ω can be decomposed as $\omega = \gamma V_L$ and the long-run variance expressed as

$$V_L = \frac{\omega}{1 - \sum_{i=1}^{q} \alpha_i - \sum_{j=1}^{p} \beta_j} \tag{2.12}$$

from which, for V_L to be defined, the following condition must be true:

$$\sum_{i=1}^{q} \alpha_i + \sum_{j=1}^{p} \beta_j < 1 \tag{2.13}$$

However, p and q values exceeding 1 are rarely encountered in practice, and thus GARCH(1,1) is by far the most popular of the GARCH models:

$$\sigma_t^2 = \omega + \alpha r_{t-1}^2 + \beta \sigma_{t-1}^2 \tag{2.14}$$

One reason for such popularity is that the GARCH(1,1) model is equivalent to

an ARCH(∞) model. Indeed, by recursive substitution,

$$\begin{aligned}
\sigma_t^2 &= \omega + \alpha r_{t-1}^2 + \beta\sigma_{t-1}^2 \\
&= \omega + \alpha r_{t-1}^2 + \beta[\omega + \alpha r_{t-2}^2 + \beta\sigma_{t-2}^2] \\
&= \omega(1+\beta) + \alpha r_{t-1}^2 + \alpha\beta r_{t-2}^2 + \beta^2\sigma_{t-2}^2 \\
&= \omega(1+\beta) + \alpha r_{t-1}^2 + \alpha\beta r_{t-2}^2 + \beta^2[\omega + \alpha r_{t-3}^2 + \beta\sigma_{t-3}^2] \\
&= \omega(1+\beta+\beta^2) + \alpha r_{t-1}^2 + \alpha\beta r_{t-2}^2 + \alpha\beta^2 r_{t-3}^2 + \beta^3\sigma_{t-3}^2 \\
&\vdots \\
&= \omega\sum_{i=0}^{\infty}\beta^i + \alpha\sum_{j=0}^{\infty}\beta^j r_{t-1-j}^2 + \lim_{j\to+\infty}\beta^j\sigma_{t-j}^2
\end{aligned}$$

since $0 > \beta < 1$ implies that $\lim_{j\to+\infty}\beta^j\sigma_{t-j}^2 = 0$ and $\sum_{i=0}^{\infty}\beta^i = \frac{1}{1-\beta}$, it follows:

$$\sigma_t^2 = \frac{\omega}{1-\beta} + \text{ARCH}(\infty) \tag{2.15}$$

where the ARCH has weights for the lagged squared returns which decay exponentially by $\alpha\sum_{j=0}^{\infty}\beta^j$.

Then, GARCH(1,1) not only results to be an ARCH model of infinite order, but it also has the advantage to be simple to use: once estimated the parameters ω, α and β, just the last squared return and the current estimate for the variance are needed to forecast the volatility one-step ahead. Notice that the EWMA model needs the same variables to be stored and it can be seen as a particular case of the GARCH(1,1) model, where $\omega = 0$, $\alpha = 1-\lambda$ and $\beta = \lambda$.

There is another useful way to re-write the GARCH(1,1) model: from Equation (2.12), setting $p, q = 1$, we have:

$$\omega = (1-\alpha-\beta)V_L \tag{2.16}$$

substituting this expression in Equation (2.14), we can reformulate the GARCH(1,1) as:

$$\sigma_t^2 = V_L + \alpha(r_{t-1}^2 - V_L) + \beta(\sigma_{t-1}^2 - V_L) \tag{2.17}$$

This means that under GARCH(1,1), the one step ahead forecast of variance corre-

sponds to the long-run variance adjusted by a term which measures the gap between the last squared return and V_L, and another term which accounts for the gap between the current estimate for the variance and V_L. Therefore, GARCH(1,1) captures the volatility mean reversion as well. This feature is even more evident when forecasting the variance $h+1$ steps ahead. Indeed,

$$\mathbb{E}_{t-1}\left[\sigma^2_{t+h}\right] = V_L + \alpha\Big(\mathbb{E}_{t-1}\left[r^2_{t+h-1}\right] - V_L\Big) + \beta\Big(\mathbb{E}_{t-1}\left[\sigma^2_{t+h-1}\right] - V_L\Big)$$

Since $\mathbb{E}_{t-1}\left[r^2_{t+h-1}\right] = \mathbb{E}_{t-1}\left[\sigma^2_{t+h-1}\right]$,

$$\mathbb{E}_{t-1}\left[\sigma^2_{t+h}\right] = V_L + (\alpha+\beta)\Big(\mathbb{E}_{t-1}\left[\sigma^2_{t+h-1}\right] - V_L\Big)$$

By recursive substitution,

$$\mathbb{E}_{t-1}\left[\sigma^2_{t+h}\right] = V_L + (\alpha+\beta)^{h+1}(\sigma^2_{t-1} - V_L) \tag{2.18}$$

Equation (2.18) implies that as the forecast horizon h grows, because $(\alpha+\beta) < 1$ the expected variance converges to its long-run average. Instead, notice that under the EWMA model, being $\alpha+\beta = 1$, the variance is expected to stay at the same level in the future. That is why the coefficient $(\alpha+\beta)$ is also called *persistence level* of the model.

2.4 Maximum likelihood estimation

Now that we have seen the most relevant econometric models to forecast volatility, in this section, we illustrate how to estimate parameters. The method used is known as *Maximum Likelihood Estimation* (MLE). The idea behind is to estimate the model's parameters in such a way that the sample would be most likely if the model was true. In other terms, it implies that once the parameters' values are known, all necessary information is available to simulate the observed random variables. This means that the joint density function of the sample data has to be known.

Since all the models covered in this chapter assume $r_t = \sigma_t z_t, \quad z_t \sim \text{IID } \mathcal{N}(0,1)$,

the following joint density function for $r_1, r_2, ..., r_t$ can be derived as follows:

$$f(r_1, r_2, ..., r_t|\boldsymbol{\theta}) = \prod_{i=1}^{t} f(r_i|\boldsymbol{\theta}) = \prod_{i=1}^{t} \left[\frac{1}{\sqrt{2\pi\sigma_i^2(\boldsymbol{\theta})}} \exp\left(-\frac{r_i^2}{2\sigma_i^2(\boldsymbol{\theta})} \right) \right] \tag{2.19}$$

Equation (2.19) expresses the joint density function of the sample conditioned on the parameter vector, $\boldsymbol{\theta}$. When interested in a function of the parameters conditioned on the data, the equation is called *likelihood function* and it is denoted as $\mathcal{L}(\boldsymbol{\theta}|r_1, r_2, ..., r_t)$. Although the two functions are the same, the likelihood function has not to be interpreted as a probability density function of the parameters: its scope is just to highlight the interest in the parameters.

Because we are interested in finding the argmax of the likelihood function and it is simpler to work with sums than products, the log of the likelihood function is usually maximized. This is possible because the natural logarithm function is monotonically increasing and so it ensures that the log of a function is maximized at the same point as the original function. The log-likelihood function can then be written as:

$$\begin{aligned} \ln \mathcal{L}(\boldsymbol{\theta}|r_1, r_2, ..., r_t) &= \ln f(r_1, r_2, ..., r_t|\boldsymbol{\theta}) = \ln \prod_{i=1}^{t} f(r_i|\boldsymbol{\theta}) = \sum_{i=1}^{t} \ln f(r_i|\boldsymbol{\theta}) \\ &= \sum_{i=1}^{t} \left[-\frac{1}{2}\ln(2\pi) - \frac{1}{2}\ln \sigma_i^2(\boldsymbol{\theta}) - \frac{r_i^2}{2\sigma_i^2(\boldsymbol{\theta})} \right] \\ &= -\frac{1}{2} \left[t \cdot \ln(2\pi) + \sum_{i=1}^{t} \ln \sigma_i^2(\boldsymbol{\theta}) + \sum_{i=1}^{t} \frac{r_i^2}{2\sigma_i^2(\boldsymbol{\theta})} \right] \end{aligned} \tag{2.20}$$

Let $\boldsymbol{\Theta}$ be a space of all the possible values of the parameters which satisfy any constraints. Estimating the parameters through MLE involves finding a unique $\hat{\boldsymbol{\theta}} \in \boldsymbol{\Theta}$ such that Equation (2.20) is maximized when $\boldsymbol{\theta} = \hat{\boldsymbol{\theta}}$. In the case of the GARCH(1,1) model, the parameter vector $\boldsymbol{\theta} \equiv [\omega, \alpha, \beta]'$ is estimated as follows:

$$\hat{\boldsymbol{\theta}}_t = \underset{\boldsymbol{\theta}\in\boldsymbol{\Theta}}{\operatorname{argmax}} \left[-\frac{t}{2}\ln(2\pi) - \frac{1}{2}\sum_{i=1}^{t} \ln\left(\omega + \alpha r_{i-1}^2 + \beta\sigma_{i-1}^2\right) - \frac{1}{2}\sum_{i=1}^{t} \frac{r_i^2}{\omega + \alpha r_{i-1}^2 + \beta\sigma_{i-1}^2} \right]$$

where σ_0^2 is initialized at $V_L = \frac{\omega}{1-\alpha-\beta}$.

This method presents several advantages: first, among all the unbiased estima-

tors it is the most efficient; second, if the likelihood function has a unique global maximum at $\boldsymbol{\theta_0}$, it is consistent, meaning that as the sample size $t \to \infty$, then $\hat{\boldsymbol{\theta}} \xrightarrow{p} \boldsymbol{\theta_0}$; third, should the joint distribution of the sample misspecified, MLE is still consistent. When this is the case, the method takes the name of Quasi Maximum Likelihood Estimation (QMLE). However, this advantage does not come without a cost: although QMLE turns out to be one of the most useful and exploited findings in modern Econometrics, it is less efficient than MLE.

Chapter 3

Machine Learning Models

3.1 Artificial neural networks

Artificial Neural Networks (ANNs) are mathematical models inspired by biological neural networks. They were first introduced by McCulloch and Pitts (1943), who presented a simplified model for nervous activity in the human brain. In recent years, as computer's processing speed and volume of available data has increased dramatically, ANNs have experienced a renewed interest within the Machine Learning field. Artificial neural networks can detect the underlying functional relationships within a set of data and perform tasks such as pattern recognition, classification and regression. Their application is ubiquitous. This chapter aims to present them as alternative tools to forecast time series, with a focus on volatility.

Compared to traditional econometric models, ANNs have, indeed, many interesting features. First, they are able to detect nonlinear structures in data without any *a priori* knowledge about the true relationships between input and output variables. Such characteristic of being non-parametric models makes them particularly suited to situations where the assumptions underlying parametric models do not adequately describe the reality. Second, the structure of ANNs can be modified to approximate a wide range of statistical and econometric models. Third, ANNs can generalize the information learned from the training set (in-sample) in order to correctly infer unseen data in the test set (out-of-sample). For all these reasons, ANNs are attractive in forecasting volatility.

The basic building block of every ANN is the artificial neuron. Similarly to its

biological counterpart, the artificial neuron receives one or more inputs and convert them in a signal. This process was modelled by McCulloch and Pitts as a weighted sum of the inputs whose output is passed through a transfer function, also called *activation function*, which according to a threshold, produces a binary output. The human brain is a collection of fully connected neurons, hence the name *neural network*. An average brain contains around 100 billion neurons. ANNs, instead, rarely exceed a few hundred or a few thousand neurons, also known as *nodes* or *processing elements (PE)*. Artificial neural networks can have different architectures, but they all arrange nodes in three different types of layers: an input layer, a hidden layer and an output layer. When the signal flow is from input to output nodes, they are called Feedforward Neural Networks (FNN). Inputs, referred to as *features*, are fed to the hidden layer and finally to the output layer where the output, referred to as *target* or *label*, is produced. ANNs can also have multiple hidden layers and when this is the case they are called Deep Neural Networks (DNNs). However, the Universal Approximation Theorem, proved by Hornik (1991), states that ANNs with a single hidden layer can approximate any continuous function arbitrarily closely. For this reason, without loss of generality, a single hidden layer network is assumed throughout this thesis. Figure 3.1 depicts an example of a feedforward neural network with m features, q hidden nodes and one output.

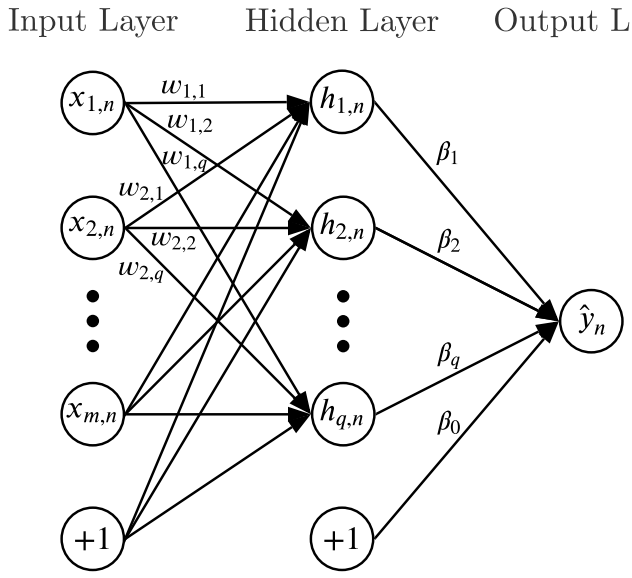

Figure 3.1: Feedforward Neural Network diagram

In order to provide the reader with a better understanding of Figure 3.1 for the case of volatility forecasting, imagine we want to predict volatility at n different time steps, each based on the volatility and squared return of the previous time step. In this case, the number of features, m, is 2 and each n-th predicted target value, $\hat{y}_n$, is produced using only the n-th set of features which are denoted by $x_{1,n}$ and $x_{2,n}$. The nodes indicating "+1" inside are meant to add a *bias* to each node in the next layer. They act as the constant term in a regression analysis. Let $\boldsymbol{x_n} = [x_{1,n}, x_{2,n}, ..., x_{m,n}, +1]^\intercal$ be the $(m+1) \times 1$ vector of input variables for the n-th time step, $\boldsymbol{h_n} = [h_{1,n}, h_{2,n}, ..., h_{q,n}]^\intercal$ the $q \times 1$ vector of hidden nodes, $\boldsymbol{\beta} = [\beta_1, \beta_2, ..., \beta_q]$ the $1 \times q$ vector of weights for the connections between the hidden and output layer, and $\boldsymbol{W}$ the $q \times (m+1)$ matrix of weights between the inputs and hidden nodes. The predicted target value, $\hat{y}_n$, can be then expressed as follows:

$$\begin{aligned} \hat{y}_n &= F\Big(\boldsymbol{\beta h_n} + \beta_0\Big) \\ \boldsymbol{h_n} &= \boldsymbol{G}\Big(\boldsymbol{W x_n}\Big) \end{aligned} \tag{3.1}$$

where F is the activation function for the output layer and $\boldsymbol{G}$ is the activation function for the hidden layer. The notation in bold for G is to denote that the function is applied element-wise to a vector or matrix.[1] F and G can be chosen among a variety of functions. The most common activation functions are: the sigmoid (or logistic) function, the hyperbolic tangent (tanh) function, and the Rectified Linear Unit (ReLU) function. Their expressions are listed below:

$$\begin{aligned} \text{sigmoid}: \quad & f(x) = \frac{1}{1+\exp^{-x}} \\ \tanh: \quad & f(x) = \frac{\exp^{x} - \exp^{-x}}{\exp^{x} + \exp^{-x}} \\ \text{ReLU}: \quad & f(x) = \max(0, x) \end{aligned}$$

Since the sigmoid function is bounded by 0 and 1, and the tanh function between -1 and 1, they are usually adopted in the hidden layer to mimic the behaviour

[1] We will use this notation throughout this thesis to refer to similar functions.

of biological neurons. The ReLU function, instead, is more used in the output layer for classification problems, while a linear function ($f(x) = x$) is preferred for regression problems. The choice of the activation functions plays an important role in the architecture of an ANN. Notice that if all the activation functions were linear, the ANN would reduce to a linear regression model, whereas non-linear functions as the sigmoid and the tanh function, allow ANNs to discover complex functional relationships that may exist between targets and features.

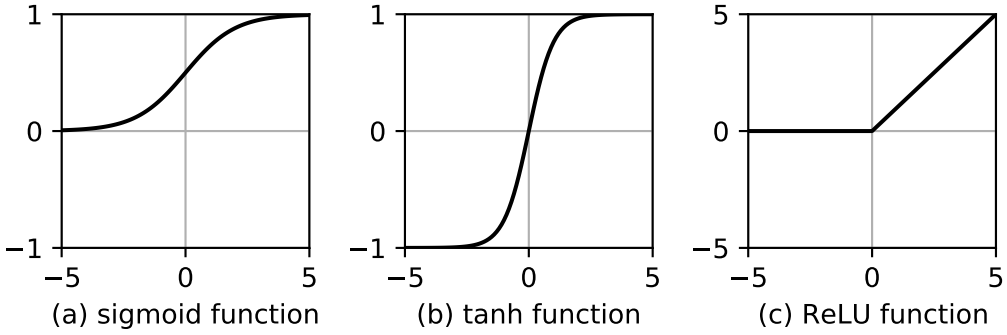

Figure 3.2: Activation Functions

3.2 Gradient descent and backpropagation

Similarly to the human brain, ANNs learn by experience. Both are able to adjust themselves in response to mistakes. However, the mechanisms involved in the learning process of the human brain are by far more complex and there is still much to be discovered, hence, what follows is limited to describe the functioning of artificial neural networks. In order to talk about mistakes, an error function, E, has to be introduced. For regression problems, like volatility forecasting, the MSE is usually chosen for this purpose:

$$E = \frac{1}{n} \sum_{n=1}^{n} (y_n - \hat{y}_n)^2 \tag{3.2}$$

The learning algorithm behind ANNs is all about finding the weights such that E is minimized. The most widely used method to deal with this numerical optimization problem is called *gradient descent*. This method iteratively seeks a minimum of a differentiable function by taking, at each iteration, a step in the opposite direction

of the gradient of the function at the current point. To put it simpler, think about a convex function, f, of a single variable, x: if $f'(x) > 0$ at a certain point, it means that $f(x)$ approximately increases if also x increases and by reducing x we get closer to a local minimum. Conversely, if $f'(x) < 0$ we must increase x for $f(x)$ to decrease. When we seek a minimum of a multi-variable function, the procedure is the same but, instead of looking just at x, all the partial derivatives are computed in order to adjust the movement on the curve with respect to multiple dimensions. The vector containing all the partial derivatives $\frac{\partial f}{\partial \boldsymbol{x}}$, with $\boldsymbol{x} = [x_1, x_2, .., x_z]$, is called gradient, and it is indicated by $\nabla f(\boldsymbol{x}) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_z}\right]$. According to the gradient descent method, after each iteration, the weights of the FNN depicted in Figure 3.1 are then adjusted as follows:

$$\Delta \boldsymbol{\beta} = -\eta \nabla E(\boldsymbol{\beta}) \tag{3.3}$$

$$\Delta \boldsymbol{w} = -\eta \nabla E(\boldsymbol{w}) \tag{3.4}$$

where $\boldsymbol{\beta}$ and $\boldsymbol{w}$ are the vectors containing all the weights between the last and first two layers, respectively, and η is a parameter called *learning rate* which controls the magnitude of the changes in weights. The learning rate is very sensitive to set because if it is too high it will overshoot the local minimum but if it is too low it will take longer to converge. Usually, a trial-and-error procedure is performed in order to set it properly.

For complex architectures of ANNs, involving many parameters, computing all the partial derivatives could be cumbersome. This is made computationally feasible by *backpropagation*. Such a method, proposed by Rumelhart et al. (1986), consists in working back through the layers calculating the partial derivatives using the chain rule. Indeed, recalling Equation (3.1), the output of an ANN is expressed as a compound function, and calculating its derivative with respect to the weights, by the chain rule, corresponds to computing the product of the derivatives of its individual functional parts. As we move backwards through the layers, the derivatives can be stored and used to calculate the new gradients, thus, avoiding redundant calculations. For example, in the case of our 3-layer FNN, both $\nabla E(\boldsymbol{\beta})$ and $\nabla E(\boldsymbol{w})$ need

the quantity $-\frac{2}{n}\sum_{n=1}^{n}(y_n - \hat{y}_n)\,F'$ to be computed, and once the result is found for $\nabla E(\boldsymbol{\beta})$, it can be stored and reused to compute $\nabla E(\boldsymbol{w})$. Of course, the advantages of the backpropagation can be better appreciated for deep neural networks, where more hidden layers and millions of weights are present.

Note that the backpropagation algorithm requires the derivative of the activation functions to exist. That is also why smooth, nonlinear activation functions, such as the sigmoid or tanh function, are used.

Although backpropagation allows to efficiently compute partial derivatives with respect to lots of weights, large data sets involving many examples in the training set could still make the learning process unfeasible. Indeed, vanilla gradient descent, also known as *batch gradient descent*, iteratively computes the gradient after each *epoch*, i.e. after all the n examples are passed through the ANN. Then, it is easy to see that, since each partial derivative of Equation (3.2) with respect to weights includes all the n sets of features, computations become more complex as n increases.

One solution to this problem is provided by the so-called *Stochastic Gradient Descent* (SGD). According to this method, instead of changing the weights after each epoch, they are adjusted after a single example. Since the gradient is calculated only on an example at a time, the steps towards the minimum of the error curve are less precise and do not necessarily let the error decrease. However, in the long run, the method succeeds in its attempt to minimize the error. Moreover, it has been recently proved by Kleinberg et al. (2018) that the noisy gradient of SGD represents an advantage with respect to vanilla gradient descent since it helps to escape from local minima and so to better minimize the error.

Another solution is provided by a mixture of batch gradient descent and SGD and it is called *mini-batch gradient descent*. As the name suggests, it is like the gradient descent but instead of having a batch including all the examples in the training set, it computes the gradient on a portion of them. Note that SGD can be seen as a particular case of mini-batch gradient descent with a *batch size* equal to 1.

3.3 Recurrent neural networks

Traditional FNNs are commonly referred to as *static neural networks* since they try to forecast target variables given sets of independent features of fixed length. They ignore the sequential order of features within each example and every new set of input variables is considered in isolation, with no memory of the previous inputs. For these reasons, feedforward neural networks are not suitable for sequential data like time series.

In response to this shortcoming, *Recurrent Neural Networks* (RNNs) were proposed by Elman (1990). Unlike FNNs, these types of ANNs allow data to propagate not only from inputs to output but also from hidden layers of earlier examples. In addition to signals from inputs, hidden nodes receive data coming from past hidden nodes as well, hence the term *recurrent*. In Figure 3.3, a diagram of the RNN is shown.

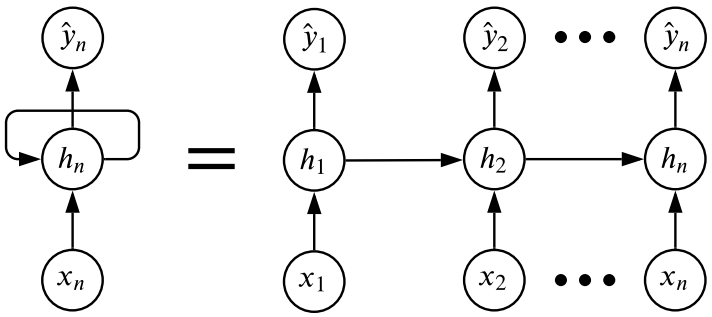

Figure 3.3: Unfolded Recurrent Neural Network diagram

Compared to Figure 3.1, here circles represents layers, and $\boldsymbol{x_n}$, $\boldsymbol{h_n}$ are vectors containing all the input and hidden nodes respectively, in the n-th example. On the right-hand side, from the unfolded representation, notice that an RNN can be seen as multiple copies of FNNs each passing a message to the successor. Such a message, represented by $\boldsymbol{h_n}$, in the context of RNNs, is also commonly referred to as *hidden state*, to emphasize its encoded and unseen nature. In this way, recurrent neural networks manage to have a memory of previous examples and detect time-dependencies in the data.

Keeping the notation used for FNNs (see Equation (3.1)), the output, $\hat{y}_n$, can be expressed as follows:

$$\begin{aligned} \hat{y}_n &= F\Big(\boldsymbol{\beta h_n} + \beta_0\Big) \\ \boldsymbol{h_n} &= \boldsymbol{G}\Big(\boldsymbol{W x_n} + \boldsymbol{U h_{n-1}}\Big) \end{aligned} \tag{3.5}$$

where $\boldsymbol{U}$ is a $q \times q$ matrix containing the weights for all the connections between $\boldsymbol{h_{n-1}}$ and $\boldsymbol{h_n}$. Another way to look at RNNs is then as FNNs with additional inputs represented by past hidden nodes.

Because of all these similarities with feedforward neural networks, RNNs can be trained by backpropagation too. The only difference is that the error is not only propagated through layers but also through time. However, although backpropagation does not pose any computational issue in RNNs, it suffers from vanishing or exploding gradient problems. Indeed, computing the gradient implies many partial derivatives to be multiplied together, and when the sequences are quite long, the product of many numbers between -1 and 1 can cause the gradient to decrease exponentially, hence to vanish. Conversely, when many partial derivatives assume absolute values greater than 1, the gradient will explode. Vanishing gradients result in the neural network to stop adjusting the weights and so to stop learning, while exploding gradients lead to over-adjustments of the weights and so to unstable weight matrices. Therefore, both these issues prevent RNNs from the capture of long-term dependencies.

3.4 Long short-term memory (LSTM)

To overcome the gradient issues associated with RNNs, Hochreiter and Schmidhuber (1997) introduced a special kind of RNN called *Long Short-Term Memory* (LSTM). This type of neural network has been so successful in detecting long-term dependencies that, at the time we write, it constitutes one of the most used models for speech recognition, language modelling, translation... etc.

Compared to RNNs, LSTMs are able to select at each time-step of the training process, what past information to forget, what new one to add and what to output.

This is accomplished through three gate mechanisms, the *forget gate*, the *input gate* and the *output gate*, which regulate the information contained in the so-called *cell state*.

To give a better idea of how the LSTM works, for the rest of this section we will implicitly refer to the diagram shown in Figure 3.4.

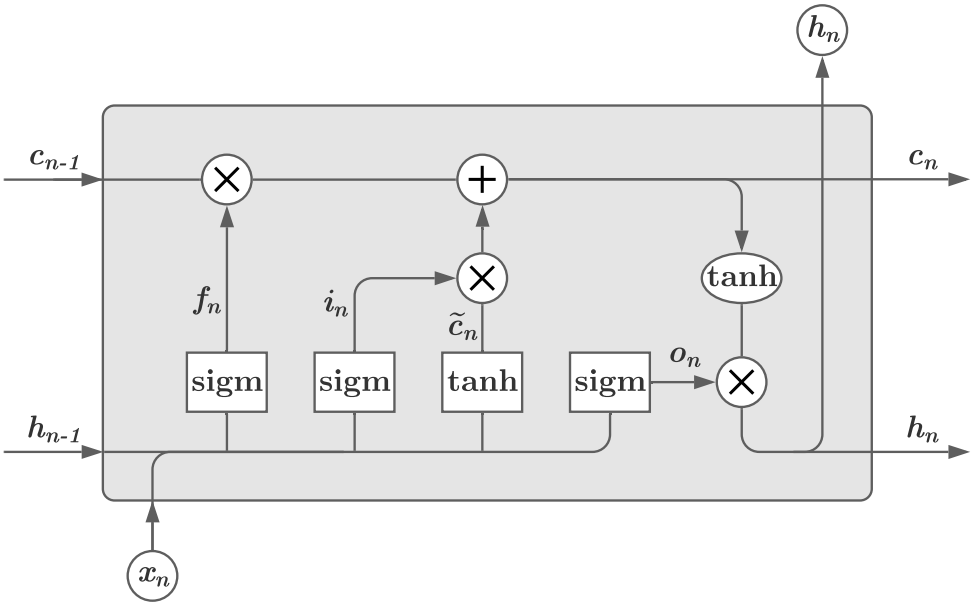

Figure 3.4: Long Short-Term Memory Diagram

Information flows both vertically from the input to the hidden layer, and horizontally from the previous to the next example. Since all the innovations to traditional RNNs are contained within the grey shaded area, referred to as *LSTM layer* or *LSTM unit*, the output layer is not shown. Rectangular shapes represent layers and the text inside indicates the applied activation function. Circles and ellipses within the LSTM unit stand for element-wise operations and annotated lines denote passing vectors. All the vectors except $\boldsymbol{x_n}$ have dimension $q \times 1$ whereas the input vector has dimension $(m+1) \times 1$ (recall that, as for FNNs, $\boldsymbol{x_n}$ includes m features and one constant, "+1", such that a bias can be added to the following layers).

The cell state, $\boldsymbol{c_n}$, is key to the LSTM unit. It acts like a conveyor belt that carries the relevant information to predict the target variable.

Starting from the left, $\boldsymbol{x_n}$ and $\boldsymbol{h_{n-1}}$ are passed through a sigmoid layer to return

the vector $\boldsymbol{f_n}$. This is what constitutes the forget gate. Since the sigmoid function outputs values in the range $[0, 1]$, elements of $\boldsymbol{f_n}$ corresponding to a value of 1 mean "keep all the information", whilst values of 0 mean "forget all this information".

$$\boldsymbol{f_n} = \mathbf{sigm}\Big(\boldsymbol{W_f x_n} + \boldsymbol{U_f h_{n-1}}\Big) \tag{3.6}$$

The next step is to decide what new information to include in the cell state. This is done through two layers that serve as the input gate. The first layer looks at the current inputs and the past hidden state and applies to their weighted sum a sigmoid function to chose which elements should be updated. The second layer, instead, applies to the same two vectors a tanh function to create new candidate elements for the cell state. The output from these two layers, $\boldsymbol{i_n}$ and $\boldsymbol{\tilde{c}_n}$ respectively, is computed as follows:

$$\boldsymbol{i_n} = \mathbf{sigm}\Big(\boldsymbol{W_i x_n} + \boldsymbol{U_i h_{n-1}}\Big) \tag{3.7}$$

$$\boldsymbol{\tilde{c}_n} = \mathbf{tanh}\Big(\boldsymbol{W_c x_n} + \boldsymbol{U_c h_{n-1}}\Big) \tag{3.8}$$

Once decided what to forget and what to add, the cell state is updated as follows:

$$\boldsymbol{c_n} = \boldsymbol{f_n} \odot \boldsymbol{c_{n-1}} + \boldsymbol{i_n} \odot \boldsymbol{\tilde{c}_n} \tag{3.9}$$

where $\odot$ denotes the Hadamard product or element-wise product.

Then, a third gate, the output gate, decides which information contained in the cell state should be output. Similarly to $\boldsymbol{f_n}$ and $\boldsymbol{i_n}$, a sigmoid layer is thus applied to $\boldsymbol{x_n}$ and $\boldsymbol{h_{n-1}}$.

$$\boldsymbol{o_n} = \mathbf{sigm}\Big(\boldsymbol{W_o x_n} + \boldsymbol{U_o h_{n-1}}\Big) \tag{3.10}$$

Before filtering the output of the LSTM unit, a final tanh function is applied to $\boldsymbol{c_n}$ in order to scale its elements between -1 and 1. The new hidden state can now be computed as follows:

$$\boldsymbol{h_n} = \boldsymbol{o_n} \odot \mathbf{tanh}\Big(\boldsymbol{c_n}\Big) \tag{3.11}$$

A copy of the vector is passed forward to the next example and the predicted target

value is computed in the same way as FNNs and RNNs:

$$\hat{y}_n = F\Big(\boldsymbol{\beta}\boldsymbol{h_n} + \beta_0\Big) \tag{3.12}$$

Chapter 4

Experimental setup and results

Following the review of the main econometric and machine learning models for volatility forecasting, in this chapter, we will assess and compare their predictive power. In particular, to avoid redundancy, since GARCH models are a generalization of ARCH models and LSTM neural networks are by construction better suited for time series than FNNs and vanilla RNNs, our experiment will be confined on comparing LSTM, GARCH(1,1) and EWMA. Because volatility is a latent variable, to assess the forecasting power of our candidate models we will use the realized volatility as a measure of the second moment of returns. For further details about the experiment, the Python code can be found in Appendix A.

4.1 Data description

Our study is conducted on the S&P 500 Index. We collect daily data about adjusted[1] close prices from Yahoo Finance between 30 December 1927 and 31 December 2020[2]. Then, the realized volatility at month t is computed as follows:

$$RV_t = \sqrt{\frac{252}{n} \sum_{i}^{n} r_{i,t}^2} \tag{4.1}$$

where $r_{i,t}$ is the i-th daily logarithmic return in month t, and n is the number of trading days within that month. Finally, also the monthly return at time t is

[1] Adjusted for dividends and stock splits
[2] Prior to March 1957 the index contains only 90 stocks

computed as:

$$r_t = \sum_{i}^{n} r_{i,t} \tag{4.2}$$

After removing any missing values, the resulting dataset is thus composed of 23361 daily returns and 1116 monthly realized volatilities and returns. A summary of statistical properties is provided in Table 4.1.

	Close price	**Daily return**	**RV**	**Monthly return**
count	23362	23361	1116	1116
mean	492.19	0.02%	15.63%	0.48%
std	737.04	1.20%	10.99%	5.40%
min	4.40	-22.90%	3.08%	-35.59%
25%	23.88	-0.45%	9.23%	-1.93%
50%	99.61	0.05%	12.46%	0.91%
75%	843.43	0.54%	17.59%	3.51%
max	3756.07	15.37%	96.55%	33.03%
skewness	1.85	-0.48	2.92	-0.62
kurtosis	5.94	22.11	14.69	10.29
ADF	4.84	-22.36^{*}	-4.82^{*}	-8.01^{*}

Note: *, ** and *** indicate significance at 1%, 5%, and 10% level, respectively.

Table 4.1: Descriptive statistics

4.2 Methodology

In order to conduct a fair comparison between the three different models, the LSTM is trained only on the monthly returns and RV series. In this way, all the models forecast one-step ahead monthly volatility conditional on the same information set.

Since the Augmented Dickey-Fuller test (ADF) rejects the null hypothesis of a unit root at the 1% level for RV and monthly returns (see Table 4.1), we do not difference these series.

We split the dataset into training, validation, and test sets, with a 6:2:2 ratio. We use the training set and the validation set to tune the hyperparameters involved in the LSTM (see Section 4.3) and then both the sets are used as the training set for the out-of-sample predictions of all the models. The time intervals for the three sets are the following:

- Training set: January 1928 - October 1983
- Validation set: November 1983 - May 2002
- Test set: June 2002 - December 2020

This framework allows us not only to validate the LSTM model on a set as large as the test set, but also to include at least one of the most extreme events inside each set. Indeed, the training set covers the years of the Great Depression, the validation set includes the Black Monday of 1987 and the test set contains the global financial crisis in 2008 and the recent COVID-19 crash. Figure 4.1 illustrates the dataset split.

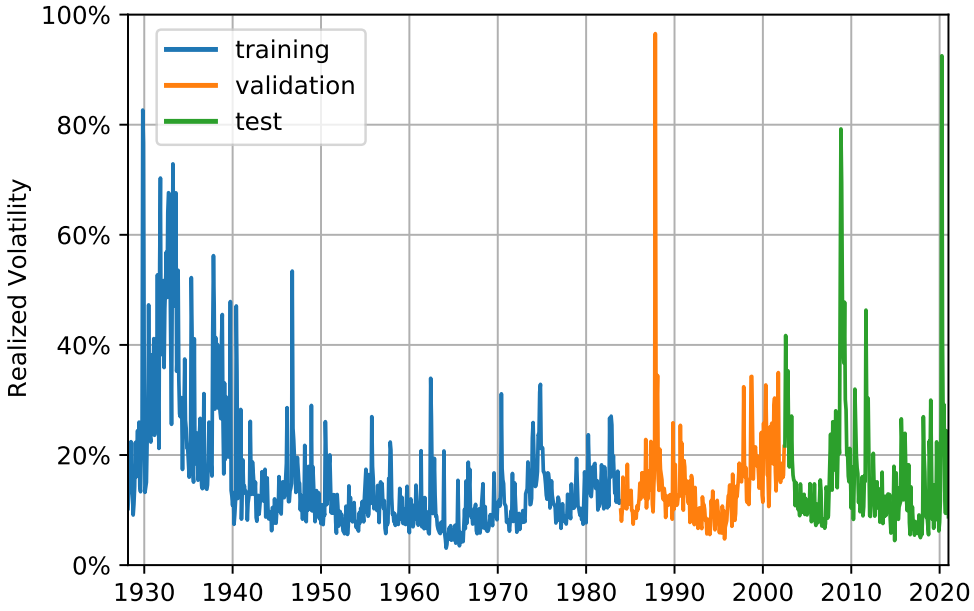

Figure 4.1: Dataset Split

Forecast accuracy is evaluated both in-sample and out-of-sample according to two performance measures: the *Root Mean Squared Error* (RMSE) and the *Mean*

Absolute Error (MAE). They are formulated as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{t}^{n}(\hat{\sigma}_t - \sigma_t)^2}$$

$$MAE = \frac{1}{n}\sum_{t}^{n}|\hat{\sigma}_t - \sigma_t|$$

Whilst the MAE measures the average magnitude of errors, each equally weighted, the RMSE gives higher weights to large errors. For such a feature, since unexpected big jumps in volatility are particularly undesirable, the LSTM is trained using the MSE as the loss function, which is more efficient to compute than the RMSE.

4.3 Hyperparameter tuning

Building the LSTM model requires the proper tuning of some hyperparameters. These are parameters that need to be specified by the researcher and which affect the learning process. It is common practice to evaluate all the possible combinations and to choose the one which performs the best in the validation set. However, since this method becomes computationally expensive as the number of parameters increases, we set up an initial architecture and then proceed to investigate the best values for a smaller set of hyperparameters.

We start by implementing an LSTM neural network in Keras[3] with a single hidden layer. A tanh activation function and a linear activation function are applied to the hidden and output layer respectively. As the optimizer, we use *Adam* (Adaptive Moment Estimation), which is a variant of the mini-batch gradient descent that adjusts the learning rate at each iteration for each model parameter.[4]

Then, the hyperparameters left to be tuned are the following:

- Number of features
- Number of hidden neurons

[3] Keras is an open-source library that provides a Python interface for developing ANNs through another library for machine learning called TensorFlow

[4] For further details, see Kingma and Ba (2014)

- Batch size
- Number of epochs

In order to speed up the research of the optimal parameter values, we first look up the best combination between the number of features and the number of hidden nodes, which are the parameters mostly affecting the model's ability to learn. Since we want the LSTM model to be trained on the realized volatility and monthly return series, the choice regarding the optimal number of features translates into how many lags to include per feature. For simplicity, we assume this number to be the same for both the features, leaving the machine to adjust the weights accordingly. Thus, we validate the model for each different combination of the number of neurons and lags per feature. We set the seed so that differences among results are not due to randomness. The batch size is set to 32 and the model is left training for a maximum of 2000 epochs, allowing an *early stopping* in case the validation error does not decrease for 100 consecutive epochs. In that case, the best weights are retrieved and the minimum error is reported. Table 4.2 shows the validation RMSE for different architectures.

No. of hidden neurons	**N. of lags per feature**					
	1	**2**	**3**	**4**	**5**	**6**
2	0.0715	**0.0709**	0.0709	0.0717	0.0724	0.0722
4	0.0726	0.0719	0.0714	0.0717	0.0708	0.0704
6	0.0727	0.0723	0.0716	0.0722	0.0725	0.0723
8	0.0729	0.0718	0.0712	0.0715	0.0720	0.0719
10	0.0730	0.0724	0.0718	0.0728	0.0735	0.0733
12	0.0736	0.0726	0.0715	0.0723	0.0728	0.0727
14	0.0733	0.0727	0.0716	0.0722	0.0728	0.0728
16	0.0731	0.0722	0.0714	0.0725	0.0731	0.0730
18	0.0726	0.0719	0.0714	0.0723	0.0729	0.0728
20	0.0731	0.0723	0.0714	0.0722	0.0728	0.0727

Table 4.2: RMSE for different combinations of hidden neurons and lags per feature

The higher the number of hidden nodes and lags per feature, the more likely the model will overfit the training data. Hence, we take into account both the RMSE and the model complexity to select the optimal combination. As a result, 2 lags per feature and 2 hidden neurons are chosen.

Once updated our initial LSTM architecture for these findings, we look into the best value for the batch size. It is common practice to use powers of 2 as the size of the batch because of better computational efficiencies.[5] In following this custom, we also add to the candidate values the case when the batch size equals 1, i.e. when it reduces to the use of SGD, and the opposite case when the batch size equals the total number of examples in the training set, which implies the use of batch gradient descent. From Figure 4.2, the optimal batch size results to be 128.

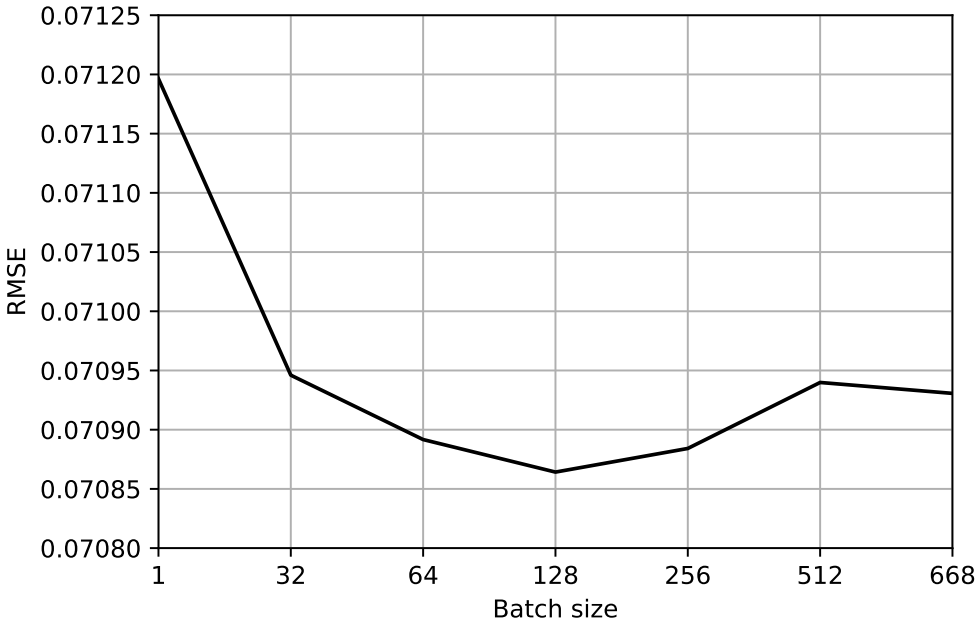

Figure 4.2: RMSE for different batch sizes

Finally, including also this finding in our LSTM model, the number of epochs is tuned. The idea, in this case, is to choose after how many epochs the LSTM should stop learning in order to perform best out-of-sample, but looking only at training and validation error. From Figure 4.3, it can be seen that starting from around 1250 epochs, both the training and validation error stabilize and the gap between the two

[5] CPUs and GPUs organize the memory in powers of 2

lines keeps relatively tight. That means the model has a good ability to generalize to new data and to not overfit when increasing the number of epochs. Hence, 1250 is the value chosen for our last hyperparameter.

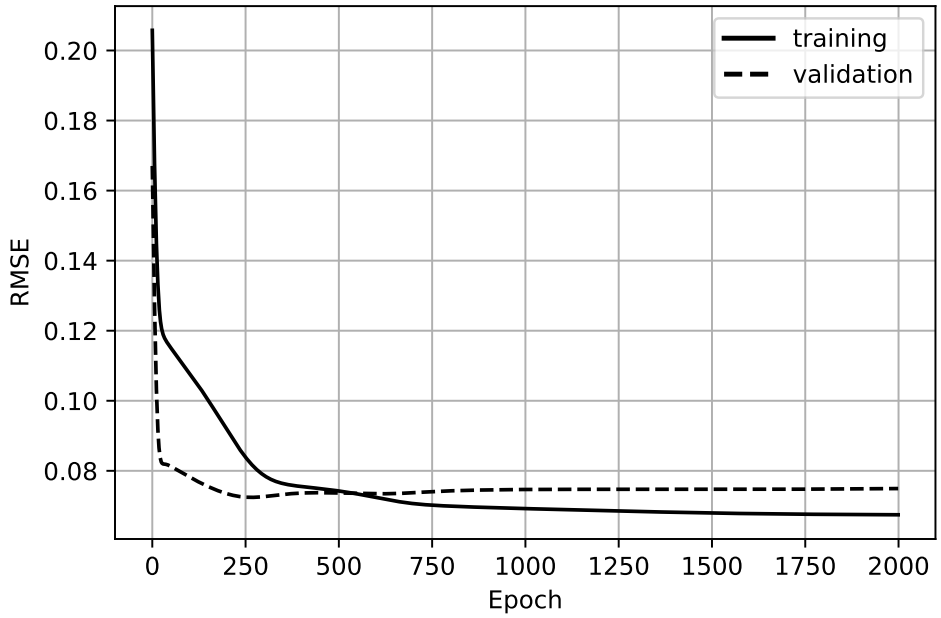

Figure 4.3: History of training and validation RMSE

4.4 Results

Once built the LSTM neural network, the three models can now be compared. Since out-of-sample forecasts are based on weights or parameters learned from data in the training and validation set, we will henceforth refer to these two sets as just the "training set".

Starting from the GARCH(1,1) and EWMA models, their respective parameter values are estimated through MLE and shown in Table 4.3.

	ω	α	β
GARCH	0.0000543	0.1246538	0.8622846
EWMA		0.1063281	0.8936719

Table 4.3: Estimated parameters for GARCH(1,1) and EWMA

The log-likelihood function is maximized through the Nelder-Mead method[6], or simplex method. The results for EWMA are expressed making use of the equivalence relations for which $\beta = \lambda$ and $\alpha = 1 - \lambda$. Applying Equation 2.12, the long-run volatility implied by the estimated GARCH(1,1) model parameters is 22.34%. Figures 4.4 and 4.5 show the fitted and predicted realized volatility by the GARCH(1,1) and EWMA model respectively.

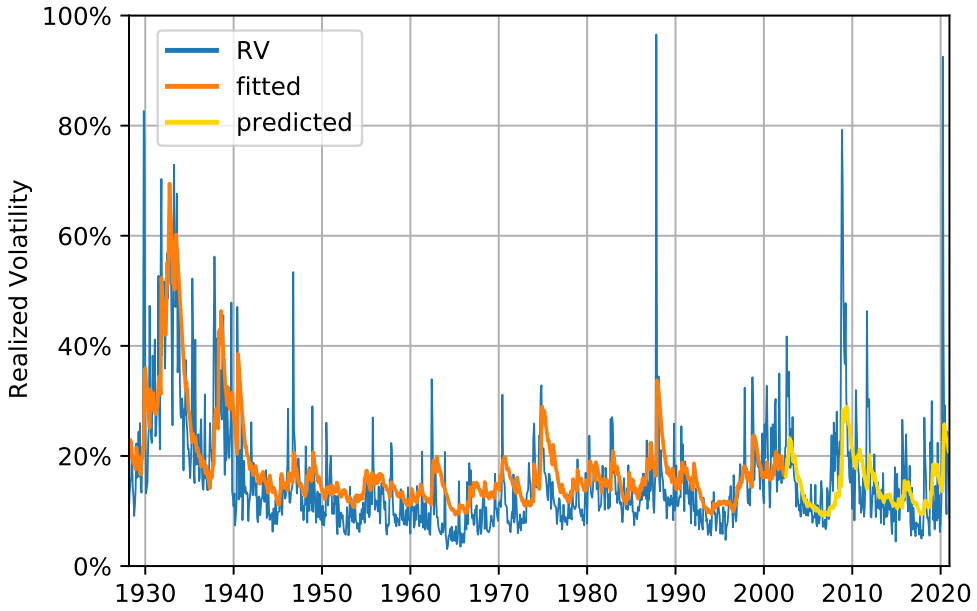

Figure 4.4: GARCH(1,1): fitted and predicted realized volatility of the S&P 500 Index

Being the estimated parameters for the two models very similar, there is no much difference between the two figures. The only source of difference is due to the push of GARCH towards the long-run volatility. However, the implied value for V_L is higher than the actual sample mean, equal to 15.63%, hence the GARCH results to have a slight upward bias in realized volatility.

Notice also that in the EWMA, the optimal value for λ is largely different from both 0.97 and 0.94, which are the values suggested by RiskMetrics for monthly and daily data respectively.

[6] For further details, see Gao and Han (2012)

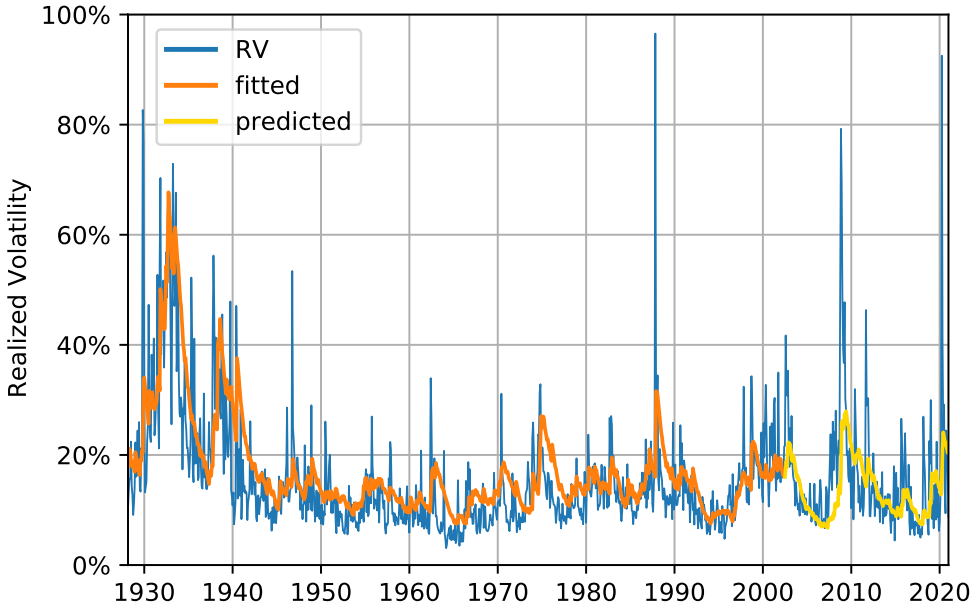

Figure 4.5: EWMA: fitted and predicted realized volatility of the S&P 500 Index

Then, in Figure 4.6, the fitted and predicted realized volatility by the LSTM neural network are shown. Two characteristics can be noted: first, the estimated volatility by the LSTM is much more flexible than the econometric models, which

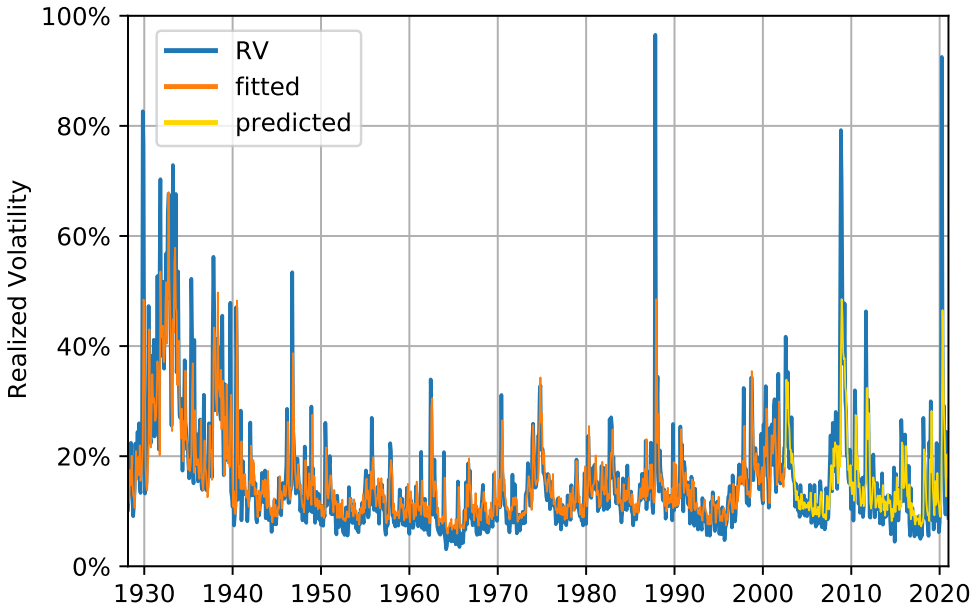

Figure 4.6: LSTM: fitted and predicted realized volatility of the S&P 500 Index

instead, tend to smooth the realized volatility; second, the LSTM has successfully learned the typical mean reverting behaviour of volatility which implies a faster convergence to the mean after shocks.

However, the model flexibility may also signal that our network is not able to generalize to out-of-sample data and that it overfits the examples in the training set. To remove any doubt and correctly compare the models, we present in Table 4.4 both their in-sample and out-of-sample RMSE and MAE.

	RMSE		**MAE**	
	In-sample	**Out-of-sample**	**In-sample**	**Out-of-sample**
EWMA	0.0786	0.1059	0.0520	0.0606
GARCH	0.0784	0.1022	0.0543	0.0598
LSTM	0.0696	0.0815	0.0411	0.0453

Table 4.4: In-sample and out-of-sample performance measures

We find that the LSTM neural network produces better results than EWMA and GARCH(1,1) both in-sample and out-of-sample. In particular, the LSTM improves the RMSE of the EWMA and GARCH models by 20.25% and 23.04% respectively, and the MAE by 24.25% and 25.24% respectively. Moreover, LSTM produces also smaller differences between out-of-sample and in-sample measures. Indeed, whilst the average differences for the econometric models correspond to 0.026 for the RMSE and 0.007 for the MAE, the LSTM yields a difference of 0.012 and 0.004 for the respective measures. That means that apart from the absolute results, the LSTM is relatively better at dealing with the problem of overfitting.

Finally, in Figure 4.7 we leave the reader with a comparison of the out-of-sample forecasts among the three different models.

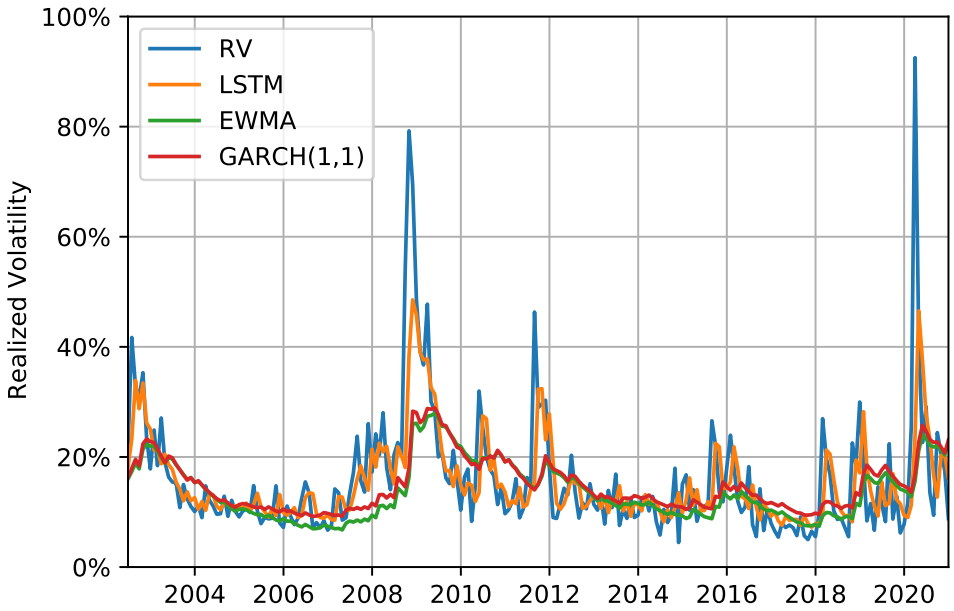

Figure 4.7: Compared out-of-sample forecasts

Conclusion

In this thesis, we compared the main econometric models for volatility forecasting to some machine learning models. In particular, our analysis covered the EWMA, ARCH and GARCH models and the FNNs, RNNs and LSTM neural networks.

A first theoretical discussion, allowed us to restrict our empirical research to the EWMA, GARCH, and LSTM models. These were compared based on their forecasting accuracy of the one-month ahead realized volatility.

We found that the LSTM neural network produces more accurate predictions, both in-sample and out-of-sample, than the econometric models considered. Moreover, the LSTM model results to better generalize to new data, hence revealing to be preferred in forecasting.

Such an outperformance may be explained by the ability of artificial neural networks to detect nonlinear relationships between input and output variables. However, the lack of a model specification, albeit attractive, does not let us catch and explain the true process governing volatility. Maybe, we must just accept that reality is too much complex to human understanding. We then let our thinking be expressed by Black (1976), who in the conclusion of his *Studies of Stock Price Volatility Changes*, states:

"Still, I don't have great confidence that I'm interpreting the results correctly. There may be other forces here that I haven't thought of. And I don't dare write down any sort of formal model of the process by which volatilities change. I'm not sure I ever will".

Bibliography

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

Andersen, T. G., & Bollerslev, T. (1998). Answering the skeptics: Yes, standard volatility models do provide accurate forecasts. *International economic review*, 885–905.

Andersen, T. G., Bollerslev, T., Diebold, F. X., & Ebens, H. (2001). The distribution of realized stock return volatility. *Journal of financial economics*, *61*(1), 43–76.

Andersen, T. G., Bollerslev, T., Diebold, F. X., & Labys, P. (2003). Modeling and forecasting realized volatility. *Econometrica*, *71*(2), 579–625.

Black, F. (1976). Studies of stock price volatility changes. *1976 Proceedings of the American Statistical Association Business and Economic Statistics Section.*

Bollerslev, T. (1986). Generalized autoregressive conditional heteroskedasticity. *Journal of econometrics*, *31*(3), 307–327.

Box, G. E., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. (2015). *Time series analysis: Forecasting and control.* Wiley.

Brownlee, J. (2017). Multivariate time series forecasting with lstms in keras. https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/

CBOE. (2019). Cboe volatility index. *White Paper*, 1–23.

Cont, R. (2001). Empirical properties of asset returns: Stylized facts and statistical issues.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, *14*(2), 179–211.

Engle, R., & Patton, A. (2001). What good is a volatility model? *Quantitative Finance*, *1*(2), 237–245.

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica: Journal of the Econometric Society*, 987–1007.

Figlewski, S. (1997). Forecasting volatility. *Financial markets, institutions & instruments*, *6*(1), 1–88.

Gao, F., & Han, L. (2012). Implementing the nelder-mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications*, *51*(1), 259–277.

Greene, W. (2018). *Econometric analysis*. Pearson.

Hewamalage, H., Bergmeir, C., & Bandara, K. (2020). Recurrent neural networks for time series forecasting: Current status and future directions. *International Journal of Forecasting*, *37*(1), 388–427.

Hinton, G. E. (1992). How neural networks learn from experience. *Scientific American*, *267*(3), 144–151.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735–1780.

Holtzman, W. H. (1950). The unbiased estimate of the population variance and standard deviation. *The American Journal of Psychology*, *63*(4), 615–617.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, *4*(2), 251–257.

Huang, W., Lai, K. K., Nakamori, Y., Wang, S., & Yu, L. (2007). Neural networks in finance and economics forecasting. *International Journal of Information Technology & Decision Making*, *6*(01), 113–140.

Hull, J. (2018). *Options, futures, and other derivatives*. Pearson.

Hull, J. (2020). *Machine learning in business: An introduction to the world of data science*. Independently Published.

Karatzas, I., & Shreve, S. E. (1988). *Brownian motion and stochastic calculus*. Springer-Verlag New York.

Kim, H. Y., & Won, C. H. (2018). Forecasting the volatility of stock price index: A hybrid model integrating lstm with multiple garch-type models. *Expert Systems with Applications*, *103*, 25–37.

Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Kleinberg, B., Li, Y., & Yuan, Y. (2018). An alternative view: When does sgd escape local minima? *International Conference on Machine Learning*, 2698–2707.

Longerstaey, J., & Spencer, M. (1996). Riskmetrics — technical document. *Morgan Guaranty Trust Company of New York*, *51*, 54.

Mandelbrot, B. (1963). The variation of certain speculative prices. *The Journal of Business*, *36*(4), 394–419.

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

Merton, R. C. (1980). On estimating the expected return on the market: An exploratory investigation. *Journal of Financial Economics*, *8*(4), 323–361.

Olah, C. (2015). Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Poon, S.-H., & Granger, C. W. (2003). Forecasting volatility in financial markets: A review. *Journal of economic literature*, *41*(2), 478–539.

Rubinstein, M. (1999). *Rubinstein on derivatives*. Risk Books.

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

Sinclair, E. (2013). *Volatility trading*. Wiley.

Tang, Z., & Fishwick, P. A. (1993). Feedforward neural nets as models for time series forecasting. *ORSA journal on computing*, *5*(4), 374–385.

Whaley, R. E. (1993). Derivatives on market volatility: Hedging tools long overdue. *The journal of Derivatives*, *1*(1), 71–84.

Whaley, R. E. (2009). Understanding the vix. *The Journal of Portfolio Management*, *35*(3), 98–105.

Appendix A

Python Code

A.1 Preliminary work

Load libraries

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import matplotlib.dates as mdates
from scipy.optimize import minimize
from keras.layers.core import Dense, Activation, Dropout
from keras.layers.recurrent import LSTM
from keras.models import Sequential
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error as mse
from sklearn.metrics import mean_absolute_error as mae
from statsmodels.stats.descriptivestats import describe
from statsmodels.tsa.stattools import adfuller as adf
import tensorflow as tf
import random as rn
import os
```

Dataset

```
# define a function to compute realized volatility from daily␣
 ↪returns
def realized_vol(x):
    return np.sqrt(sum(x**2)*252/(len(x)))

# load the daily data
data = pd.read_csv('SPX.csv', parse_dates=True)
# daily log returns
data['SPX_log_ret'] = np.log(data['Adj Close']).diff()
# convert 'Date' in datetime format
data['Date']=data['Date'].astype('datetime64[ns]')

# monthly realized volatility
RV_m = data.resample('M', on='Date')['SPX_log_ret'].
 ↪apply(realized_vol)
# monthly returns
ret_m = data.resample('M', on='Date')['SPX_log_ret'].sum()

# concatenate data
data_m=pd.concat([RV_m,ret_m],axis=1)
data_m.columns = ['RV_m','ret_m']

# drop NaN values
data_m.dropna(axis=0,inplace=True)

# export dataset description to Excel
data_desc = pd.concat([describe(data),describe(data_m)],axis=1)
data_desc.to_excel("data_description.xlsx")
```

```
# compute Augmented Dickey-Fuller test
for i in [data['Adj Close'],data['SPX_log_ret'].iloc[1:
 ↪],data_m['RV_m'],data_m['ret_m']]:
    result = adf(i)
    print('ADF stat: %.2f' % result[0],'p-value: %f' % result[1])

def perf_measures(x, x_hat,lags,row_split,row_name):
    rmse_in = np.sqrt(mse(x.iloc[lags:row_split], x_hat.iloc[lags:
 ↪row_split]))
    rmse_out = np.sqrt(mse(x.iloc[row_split:], x_hat.
 ↪iloc[row_split:]))
    mae_in = mae(x.iloc[lags:row_split], x_hat.iloc[lags:
 ↪row_split])
    mae_out = mae(x.iloc[row_split:], x_hat.iloc[row_split:])
    data={'RMSE_in': rmse_in, 'RMSE_out': rmse_out,'MAE_in':␣
 ↪mae_in, 'MAE_out': mae_out}
    output = pd.DataFrame(data,index = [row_name])
    return output
```

A.2 LSTM

Auxiliary function

```
# convert series to supervised learning (Source:␣
 ↪machinelearningmastery.com)
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = pd.DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
```

```
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in␣
↪range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in␣
↪range(n_vars)]
    # put it all together
    agg = pd.concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg
```

Main function

```
def LSTM_model(neurons,lags,batch_size, epochs=1000,␣
↪validation=False, early_stop=False):


    #---- The following part is necessary to ---
    #---- get reproducible results in Keras ----


    seed_value=1234
    os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
```

```
    os.environ["CUDA_VISIBLE_DEVICES"] = ""
    os.environ['PYTHONHASHSEED']=str(seed_value)

    np.random.seed(seed_value)
    rn.seed(seed_value)
    tf.compat.v1.set_random_seed(seed_value)

    session_conf = tf.compat.v1.
↪ConfigProto(intra_op_parallelism_threads=1,␣
↪inter_op_parallelism_threads=1)
    sess = tf.compat.v1.Session(graph=tf.compat.v1.
↪get_default_graph(), config=session_conf)
    tf.compat.v1.keras.backend.set_session(sess)
    #-----------------------------------------

    values = data_m[['RV_m','ret_m']].values

    # normalize features
    scaler = MinMaxScaler(feature_range=(-1, 1))
    scaled = scaler.fit_transform(values)

    # number of lags per feature
    timesteps = lags
    # number of features
    features = 2

    # create examples from our data set to be fed into the LSTM
    reframed = series_to_supervised(scaled, timesteps, 1)
    values = reframed.values
```

```
    # split the dataset
    train_row = int(round(0.60 * reframed.shape[0]))
    valid_row = int(round(0.80 * reframed.shape[0]))
    if validation == True:
        train = values[:train_row, :]
        test = values[train_row:valid_row,:]
    else:
        train = values[:valid_row, :]
        test = values[valid_row:,:]

    # split into input and outputs
    train_X, train_y = train[:, :(timesteps * features)], train[:,␣
↪-features]
    test_X, test_y = test[:, :(timesteps * features)], test[:,␣
↪-features]

    # reshape input to be 3D [samples, timesteps, features]
    train_X = train_X.reshape(train_X.shape[0], timesteps,␣
↪features)
    test_X = test_X.reshape(test_X.shape[0], timesteps, features)

    # design network
    model = Sequential()
    model.add(LSTM(neurons, input_shape=(train_X.shape[1], train_X.
↪shape[2]), activation='tanh'))
    model.add(Dense(1, activation='linear'))
    model.compile(loss='mse', optimizer='adam')

    # early stopping
    if early_stop == True:
```

```
        early_stopping = [EarlyStopping(monitor="val_loss",␣
↪verbose=0, mode='min', patience=100, restore_best_weights=True)]
    else:
        early_stopping = False
    # learning process
    learning = model.fit(train_X, train_y, epochs=epochs,␣
↪batch_size=batch_size, validation_data=(test_X,test_y),␣
↪verbose=0, shuffle=False, callbacks=early_stopping)

    # define a function to return predictions from LSTM
    def LSTM_predict(model, X, y):
        # make a prediction
        yhat = model.predict(X)
        X = X.reshape((X.shape[0], features*timesteps))
        # invert scaling for forecast
        inv_yhat = np.concatenate((yhat, X[:, -1:]), axis=1)
        inv_yhat = scaler.inverse_transform(inv_yhat)
        inv_yhat = inv_yhat[:,0]
        # invert scaling for actual
        y = y.reshape((len(y), 1))
        inv_y = np.concatenate((y, X[:, -1:]), axis=1)
        inv_y = scaler.inverse_transform(inv_y)
        inv_y = inv_y[:,0]
        return pd.DataFrame({'RV': np.array(inv_y), 'LSTM': np.
↪array(inv_yhat)})

    # predicted values in-sample and out-of-sample
    LSTM_train = LSTM_predict(model, train_X, train_y)
    LSTM_test = LSTM_predict(model, test_X, test_y)
```

```
    # compute performance measures in-sample and out-of-sample
    rmse_in = np.sqrt(mse(LSTM_train['RV'], LSTM_train['LSTM']))
    rmse_out = np.sqrt(mse(LSTM_test['RV'], LSTM_test['LSTM']))
    mae_in = mae(LSTM_train['RV'], LSTM_train['LSTM'])
    mae_out = mae(LSTM_test['RV'], LSTM_test['LSTM'])

    data = {'RMSE_in': rmse_in, 'RMSE_out': rmse_out,'MAE_in':␣
↪mae_in, 'MAE_out': mae_out}
    perf_meas = pd.DataFrame(data,index = ['LSTM'])

    yhat = np.concatenate([np.full((lags,), np.
↪nan),LSTM_train['LSTM'].values,LSTM_test['LSTM'].values],axis=0)

    class output:
        def __init__(self):
            self.learning = learning # data about learning
            self.perf_meas = perf_meas # performance measures
            self.yhat = yhat # predicted values
    return output()
```

Hyperparameter Optimization

```
###### find the optimal number of neurons and lags ######

n_lags = []
n_neurons = []
rmse = []
for lags in np.arange(1,7,1):
    for neuron in np.arange(2,22,2):
        n_lags.append(lags)
```

```
        n_neurons.append(neuron)
        output = LSTM_model(neuron, lags, batch_size=32,␣
 ↪epochs=2000, validation=True, early_stop=True).
 ↪perf_meas['RMSE_out'][0]
        rmse.append(output)
        print(lags,neuron,output)
hyp_1 = pd.DataFrame({'lags': n_lags,'neurons': n_neurons,'rmse':
 ↪rmse})
hyp_1.to_excel("table_neurons-lags.xlsx")
```

```
######## find the best batch size ########


lags = 2
neurons = 2


batchsize = []
rmse = []
bs_list = [int(round(2**i)) for i in np.arange(5,10,1)]
# stochastic gradient descent
bs_list.insert(0, 1)
# batch gradient descent
n = int(round(0.60 * (data_m.shape[0]-lags)))
bs_list.append(n)
for bs in bs_list:
    batchsize.append(bs)
    output = LSTM_model(neurons, lags, batch_size=bs, epochs=2000,␣
 ↪validation=True, early_stop=True).perf_meas['RMSE_out'][0]
    rmse.append(output)
    print(bs,output)
hyp_2 = pd.DataFrame({'batch_size': batchsize,'rmse':rmse})
hyp_2.to_excel("table_batch_sizes.xlsx")
```

```
#plot RMSE vs batch size
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(rmse, 'k-')
# aesthetic adjustments
ax.grid()
ax.set_xticks(np.arange(0,len(batchsize),1))
numbers_str = [str(i) for i in batchsize]
ax.set_xticklabels(numbers_str)
ax.set_xlim(0,6)
ax.set_ylim(0.07080,0.07125)
ax.set(xlabel='Batch size', ylabel='RMSE')
plt.subplots_adjust(left=0.15)
fig.savefig("Best_batch_size.pdf")
plt.show()
```

```
######## find the best epoch ########

best_LSTM = LSTM_model(neurons, lags, batch_size=128, epochs=2000,␣
 ↪validation=True, early_stop=False)

loss_train = best_LSTM.learning.history['loss']
loss_valid = best_LSTM.learning.history['val_loss']

# plot train error vs test error
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.plot(np.sqrt(loss_train), 'k-', label='training')
ax.plot(np.sqrt(loss_valid), 'k--',label='validation')
# aesthetic adjustments
```

```
ax.grid()
leg = ax.legend(loc="best")
for line in leg.get_lines():
    line.set_linewidth(2)
ax.set(xlabel='Epoch', ylabel='RMSE')
fig.savefig("Best_epoch.pdf")
plt.show()
```

```
######## plot the dataset split ########

train_row = int(round(0.60 * (data_m.shape[0]-lags))) + lags
valid_row = int(round(0.80 * (data_m.shape[0]-lags))) + lags

# create figure
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.xaxis.set_major_locator(mdates.YearLocator(10))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
ax.plot(data_m['RV_m'].iloc[lags:train_row],label='training')
ax.plot(data_m['RV_m'].iloc[train_row:valid_row],␣
 ↪label='validation')
ax.plot(data_m['RV_m'].iloc[valid_row:], label='test')
# aesthetic adjustments
ax.grid()
leg = ax.legend(loc="upper left", bbox_to_anchor=(0.02, 0.995))
for line in leg.get_lines():
    line.set_linewidth(2)
datemin = np.datetime64(data_m.index[lags], 'D')
datemax = np.datetime64(data_m.index[-1])
ax.set_xlim(datemin,datemax)
ax.set_ylim(0,1)
```

```
ax.yaxis.set_major_locator(mtick.FixedLocator(ax.get_yticks()))
ax.set_yticklabels(['{:.0f}%'.format(y*100) for y in ax.
 ↪get_yticks()])
plt.subplots_adjust(bottom=0.1)
ax.set_ylabel('Realized Volatility')
fig.savefig("Dataset_split.pdf")
plt.show()
```

Out-of-sample forecasting

```
best_LSTM = LSTM_model(neurons=2, lags=2, batch_size=128,␣
 ↪epochs=1250, validation=False, early_stop=False)
data_m['LSTM'] = best_LSTM.yhat


# create figure
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.xaxis.set_major_locator(mdates.YearLocator(10))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
ax.plot(data_m['RV_m'].iloc[lags:],label='RV')
ax.plot(data_m['LSTM'].iloc[lags:valid_row],lw=0.75,␣
 ↪label='fitted')
ax.plot(data_m['LSTM'].iloc[valid_row:],'gold',lw=0.75,␣
 ↪label='predicted')
# aesthetic adjustments
ax.grid()
leg = ax.legend(loc="upper left", bbox_to_anchor=(0.02, 0.995))
for line in leg.get_lines():
    line.set_linewidth(2)
datemin = np.datetime64(data_m.index[lags], 'D')
```

```
datemax = np.datetime64(data_m.index[-1])
ax.set_xlim(datemin,datemax)
ax.set_ylim(0,1)
ax.yaxis.set_major_locator(mtick.FixedLocator(ax.get_yticks()))
ax.set_yticklabels(['{:.0f}%'.format(y*100) for y in ax.
 ↪get_yticks()])
plt.subplots_adjust(bottom=0.1)
ax.set_ylabel('Realized Volatility')
fig.savefig("LSTM_train-test.pdf")
plt.show()

# show performance measures
pm_LSTM = best_LSTM.perf_meas
pm_LSTM
```

A.3 GARCH(1,1)

```
# define a function to create a GARCH(1,1) process
def garch_fct(omega, alpha, beta, rets):
    T = len(rets)
    sigma2 = np.zeros(T)
    for i in range(T):
        if i==0:
            sigma2[i] = omega/(1-alpha-beta)
        else:
            sigma2[i] = omega + alpha*rets[i-1]**2+beta*sigma2[i-1]
    return sigma2

# define a function to compute the negative log-likelihood
def garch_loglik(param,rets):
```

```
    T = len(rets)
    omega=param[0]
    alpha=param[1]
    beta=param[2]

    sigma2 = garch_fct(omega, alpha, beta, rets)
    loglik = np.sum(-np.log(sigma2)-rets**2/sigma2)

    return -loglik

# initial parameters
param0 = (0.00001, 0.01, 0.9)
# log-returns
rets = data_m['ret_m'].values
result = minimize(garch_loglik, param0, args=rets,␣
 ↪method='nelder-mead', options={'disp':True})

omega=result.x[0]
alpha=result.x[1]
beta=result.x[2]

data_m['GARCH'] = np.sqrt(garch_fct(omega, alpha, beta, rets)*12)

#create figure
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.xaxis.set_major_locator(mdates.YearLocator(10))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
ax.plot(data_m['RV_m'].iloc[lags:],lw=0.75,label='RV')
ax.plot(data_m['GARCH'].iloc[lags:valid_row],label='fitted')
```

```
ax.plot(data_m['GARCH'].iloc[valid_row:],'gold',label='predicted')
# aesthetic adjustments
ax.grid()
leg = ax.legend(loc="upper left", bbox_to_anchor=(0.02, 0.995))
for line in leg.get_lines():
    line.set_linewidth(2)
datemin = np.datetime64(data_m.index[lags], 'D')
datemax = np.datetime64(data_m.index[-1])
ax.set_xlim(datemin,datemax)
ax.set_ylim(0,1)
ax.yaxis.set_major_locator(mtick.FixedLocator(ax.get_yticks()))
ax.set_yticklabels(['{:.0f}%'.format(y*100) for y in ax.
 ↪get_yticks()])
plt.subplots_adjust(bottom=0.1)
ax.set_ylabel('Realized Volatility')
fig.savefig("GARCH.pdf")
plt.show()

# show results and performance measures
print(omega,alpha,beta)
V_L = np.sqrt((omega/(1-alpha-beta))*12)
print('Long-run volatility: %.4f' % V_L)
pm_GARCH = perf_measures(data_m['RV_m'],␣
 ↪data_m['GARCH'],2,valid_row,'GARCH')
pm_GARCH
```

A.4 EWMA

```
# define a function to create a EWMA process
def EWMA_fct(lam, rets):
    T = len(rets)
    sigma2 = np.zeros(T)
    for i in range(T):
        if i==0:
            sigma2[i] = (0.2**2)/12
        else:
            sigma2[i] = (1-lam)*rets[i-1]**2 + lam*sigma2[i-1]
    return sigma2

# define a function to compute the negative log-likelihood
def EWMA_loglik(lam,rets):
    T = len(rets)

    sigma2 = EWMA_fct(lam, rets)
    loglik = np.sum(-np.log(sigma2)-rets**2/sigma2)

    return -loglik

result = minimize(EWMA_loglik, 0.90, args=rets,␣
 ↪method='nelder-mead', options={'disp':True})

# lambda value
lam = result.x[0]

data_m['EWMA'] = np.sqrt(EWMA_fct(lam, rets)*12)
```

```
#create figure
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.xaxis.set_major_locator(mdates.YearLocator(10))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
ax.plot(data_m['RV_m'].iloc[lags:],lw=0.75,label='RV')
ax.plot(data_m['EWMA'].iloc[lags:valid_row],label='fitted')
ax.plot(data_m['EWMA'].iloc[valid_row:],'gold',label='predicted')
# aesthetic adjustments
ax.grid()
leg = ax.legend(loc="upper left", bbox_to_anchor=(0.02, 0.995))
for line in leg.get_lines():
    line.set_linewidth(2)
datemin = np.datetime64(data_m.index[lags], 'D')
datemax = np.datetime64(data_m.index[-1])
ax.set_xlim(datemin,datemax)
ax.set_ylim(0,1)
ax.yaxis.set_major_locator(mtick.FixedLocator(ax.get_yticks()))
ax.set_yticklabels(['{:.0f}%'.format(y*100) for y in ax.
 ↪get_yticks()])
plt.subplots_adjust(bottom=0.1)
ax.set_ylabel('Realized Volatility')
fig.savefig("EWMA.pdf")
plt.show()


# show results and performance measures
print('Lambda: %.7f' % lam)
pm_EWMA = perf_measures(data_m['RV_m'],␣
 ↪data_m['EWMA'],2,valid_row,'EWMA')
pm_EWMA
```

A.5 Comparing the models

```
#create figure
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.xaxis.set_major_locator(mdates.YearLocator(2))
ax.xaxis.set_major_formatter(mdates.DateFormatter('%Y'))
ax.plot(data_m['RV_m'].iloc[valid_row:],label='RV')
ax.plot(data_m['LSTM'].iloc[valid_row:],label='LSTM')
ax.plot(data_m['EWMA'].iloc[valid_row:], label='EWMA')
ax.plot(data_m['GARCH'].iloc[valid_row:], label='GARCH(1,1)')
# aesthetic adjustments
ax.grid()
datemin = np.datetime64(data_m.index[valid_row], 'D')
datemax = np.datetime64(data_m.index[-1])
ax.set_xlim(datemin,datemax)
ax.yaxis.set_major_locator(mtick.FixedLocator(ax.get_yticks()))
ax.set_yticklabels(['{:.0f}%'.format(y*100) for y in ax.
 ↪get_yticks()])
ax.set_ylim(0,1)
ax.legend(loc="best")
plt.subplots_adjust(bottom=0.1)
ax.set_ylabel('Realized Volatility')
fig.savefig("Summary.pdf")
plt.show()


# show and export performance measures to Excel
summary = pd.concat([pm_EWMA,pm_GARCH,pm_LSTM],axis=0)
summary.to_excel("summary_results.xlsx")
summary
```

Summary

Volatility is a central topic in the financial literature and such paramount importance lies in the vast array of its applications.

For example, volatility represents an essential element to many investment decisions as it is often taken as the starting point for optimal portfolio allocations. In 1952, Harry Markowitz laid the foundation of the Modern Portfolio Theory by which investors are risk averse and have a utility function increasing with expected return and decreasing with volatility.

Volatility is also a key input in the pricing of many derivatives. Indeed, to evaluate options' fair value, the volatility of the underlying asset until the expiration date must be known. Besides, in recent years, even derivatives with volatility itself as the underlying have been introduced, and in these cases, the definition and measurement of volatility must be specified in the derivative contracts.

In risk management, volatility is relevant for computing the Value at Risk (VaR), whose estimation has become a standard practice for financial institutions. Indeed, since the first Basel Accord was established in 1996, banks and trading venues are required to set aside a reserve capital of at least three times that of VaR.

Volatility can also have wide consequences on the economy as a whole. Thus, policymakers regard volatility as an indicator of uncertainty in the financial market. For example, both the Federal Reserve and the Bank of England take into account the securities volatility in establishing their monetary policies. Besides, volatility negatively affects market liquidity since when the former spikes, the latter usually declines.

Volatility is crucial for hedging strategies as well. Indeed, during stressed market conditions, not only volatility increases but also correlations among different secu-

rities do so. In these circumstances, derivative instruments may work as insurance against sudden market downturns.

For all these reasons, the relevance of volatility forecasting follows as a natural consequence. Although the literature on this subject is extensive and many models have been proposed, in the last years, we have been assisting to a rise in the applications of machine learning techniques to many different sectors. Thus, this thesis is aimed at bridging the gap between the classical financial literature and the machine learning models for volatility forecasting.

In particular, we compare the predictive power of the Exponential Weighted Moving Average (EWMA) model and the Generalized AutoRegressive Conditional Heteroscedasticity (GARCH) model with a Long Short Term Memory (LSTM) neural network.

Our study is conducted on the S&P 500 Index. We collect daily data about adjusted[1] close prices from Yahoo Finance between 30 December 1927 and 31 December 2020[2]. Then, the realized volatility at month t is computed as follows:

$$RV_t = \sqrt{\frac{252}{n}\sum_{i}^{n} r_{i,t}^2} \tag{1}$$

where $r_{i,t}$ is the i-th daily logarithmic return in month t, and n is the number of trading days within that month. Finally, also the monthly return at time t is computed as:

$$r_t = \sum_{i}^{n} r_{i,t} \tag{2}$$

After removing any missing values, the resulting dataset is thus composed of 23361 daily returns and 1116 monthly realized volatilities and returns. A summary of statistical properties is provided in Table 1.

[1] Adjusted for dividends and stock splits
[2] Prior to March 1957 the index contains only 90 stocks

	Close price	Daily return	RV	Monthly return
count	23362	23361	1116	1116
mean	492.19	0.02%	15.63%	0.48%
std	737.04	1.20%	10.99%	5.40%
min	4.40	-22.90%	3.08%	-35.59%
25%	23.88	-0.45%	9.23%	-1.93%
50%	99.61	0.05%	12.46%	0.91%
75%	843.43	0.54%	17.59%	3.51%
max	3756.07	15.37%	96.55%	33.03%
skewness	1.85	-0.48	2.92	-0.62
kurtosis	5.94	22.11	14.69	10.29
ADF	4.84	-22.36^{*}	-4.82^{*}	-8.01^{*}

Note: *, ** and *** indicate significance at 1%, 5%, and 10% level, respectively.

Table 1: Descriptive statistics

In order to conduct a fair comparison between the three different models, the LSTM is trained only on the monthly returns and RV series. In this way, all the models forecast one-step ahead monthly volatility conditional on the same information set.

Since the Augmented Dickey-Fuller test (ADF) rejects the null hypothesis of a unit root at the 1% level for RV and monthly returns (see Table 1), we do not difference these series.

We split the dataset into training, validation, and test sets, with a 6:2:2 ratio. We use the training set and the validation set to tune the hyperparameters involved in the LSTM and then both the sets are used as the training set for the out-of-sample predictions of all the models. The time intervals for the three sets are the following:

- Training set: January 1928 - October 1983
- Validation set: November 1983 - May 2002
- Test set: June 2002 - December 2020

This framework allows us not only to validate the LSTM model on a set as large as the test set, but also to include at least one of the most extreme events inside each set. Indeed, the training set covers the years of the Great Depression, the validation set includes the Black Monday of 1987 and the test set contains the global financial crisis in 2008 and the recent COVID-19 crash. Figure 1 illustrates the dataset split.

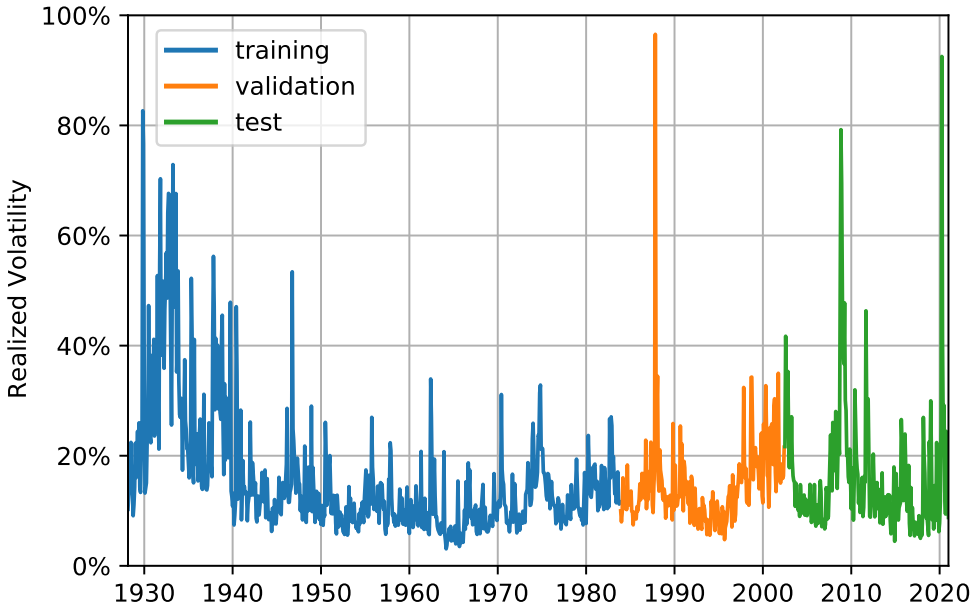

Figure 1: Dataset Split

Forecast accuracy is evaluated both in-sample and out-of-sample according to two performance measures: the *Root Mean Squared Error* (RMSE) and the *Mean Absolute Error* (MAE). They are formulated as follows:

$$RMSE = \sqrt{\frac{1}{n}\sum_{t}^{n}(\hat{\sigma}_t - \sigma_t)^2}$$

$$MAE = \frac{1}{n}\sum_{t}^{n}|\hat{\sigma}_t - \sigma_t|$$

Whilst the MAE measures the average magnitude of errors, each equally weighted, the RMSE gives higher weights to large errors. For such a feature, since unexpected big jumps in volatility are particularly undesirable, the LSTM is trained using the MSE as the loss function, which is more efficient to compute than the RMSE.

Building the LSTM model requires the proper tuning of some hyperparameters. These are parameters that need to be specified by the researcher and which affect the learning process. It is common practice to evaluate all the possible combinations and to choose the one which performs the best in the validation set. However, since this method becomes computationally expensive as the number of parameters increases, we set up an initial architecture and then proceed to investigate the best values for a smaller set of hyperparameters.

We start by implementing an LSTM neural network in Keras[3] with a single hidden layer. A tanh activation function and a linear activation function are applied to the hidden and output layer respectively. As the optimizer, we use *Adam* (Adaptive Moment Estimation), which is a variant of the mini-batch gradient descent that adjusts the learning rate at each iteration for each model parameter.[4]

Then, the hyperparameters left to be tuned are the following:

- Number of features
- Number of hidden neurons
- Batch size
- Number of epochs

In order to speed up the research of the optimal parameter values, we first look up the best combination between the number of features and the number of hidden nodes, which are the parameters mostly affecting the model's ability to learn. Since we want the LSTM model to be trained on the realized volatility and monthly return series, the choice regarding the optimal number of features translates into how many lags to include per feature. For simplicity, we assume this number to be the same for both the features, leaving the machine to adjust the weights accordingly. Thus, we validate the model for each different combination of the number of neurons and lags per feature. We set the seed so that differences among results are not due to randomness. The batch size is set to 32 and the model is left training for a

[3] Keras is an open-source library that provides a Python interface for developing ANNs through another library for machine learning called TensorFlow

[4] For further details, see Kingma and Ba (2014)

maximum of 2000 epochs, allowing an *early stopping* in case the validation error does not decrease for 100 consecutive epochs. In that case, the best weights are retrieved and the minimum error is reported. Table 2 shows the validation RMSE for different architectures.

No. of hidden neurons	**N. of lags per feature**					
	1	**2**	**3**	**4**	**5**	**6**
2	0.0715	**0.0709**	0.0709	0.0717	0.0724	0.0722
4	0.0726	0.0719	0.0714	0.0717	0.0708	0.0704
6	0.0727	0.0723	0.0716	0.0722	0.0725	0.0723
8	0.0729	0.0718	0.0712	0.0715	0.0720	0.0719
10	0.0730	0.0724	0.0718	0.0728	0.0735	0.0733
12	0.0736	0.0726	0.0715	0.0723	0.0728	0.0727
14	0.0733	0.0727	0.0716	0.0722	0.0728	0.0728
16	0.0731	0.0722	0.0714	0.0725	0.0731	0.0730
18	0.0726	0.0719	0.0714	0.0723	0.0729	0.0728
20	0.0731	0.0723	0.0714	0.0722	0.0728	0.0727

Table 2: RMSE for different combinations of hidden neurons and lags per feature

The higher the number of hidden nodes and lags per feature, the more likely the model will overfit the training data. Hence, we take into account both the RMSE and the model complexity to select the optimal combination. As a result, 2 lags per feature and 2 hidden neurons are chosen.

Once updated our initial LSTM architecture for these findings, we look into the best value for the batch size. It is common practice to use powers of 2 as the size of the batch because of better computational efficiencies.[5] In following this custom, we also add to the candidate values the case when the batch size equals 1, i.e. when it reduces to the use of SGD, and the opposite case when the batch size equals the total number of examples in the training set, which implies the use of batch gradient descent. From Figure 2, the optimal batch size results to be 128.

[5] CPUs and GPUs organize the memory in powers of 2

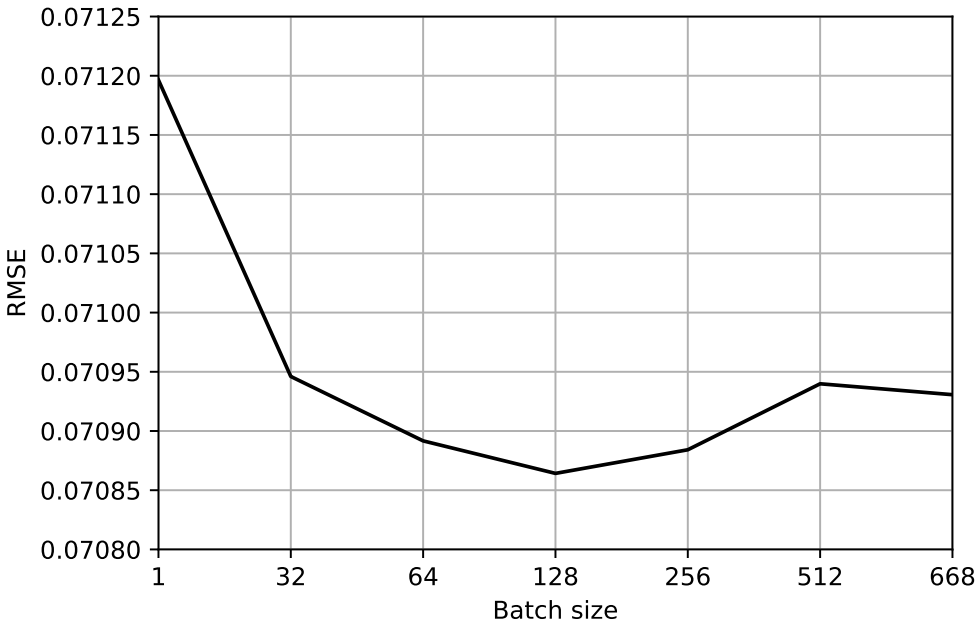

Figure 2: RMSE for different batch sizes

Finally, including also this finding in our LSTM model, the number of epochs is tuned. The idea, in this case, is to choose after how many epochs the LSTM should stop learning in order to perform best out-of-sample, but looking only at training and validation error. From Figure 3, it can be seen that starting from around 1250 epochs, both the training and validation error stabilize and the gap between the two lines keeps relatively tight. That means the model has a good ability to generalize to new data and to not overfit when increasing the number of epochs. Hence, 1250 is the value chosen for our last hyperparameter.

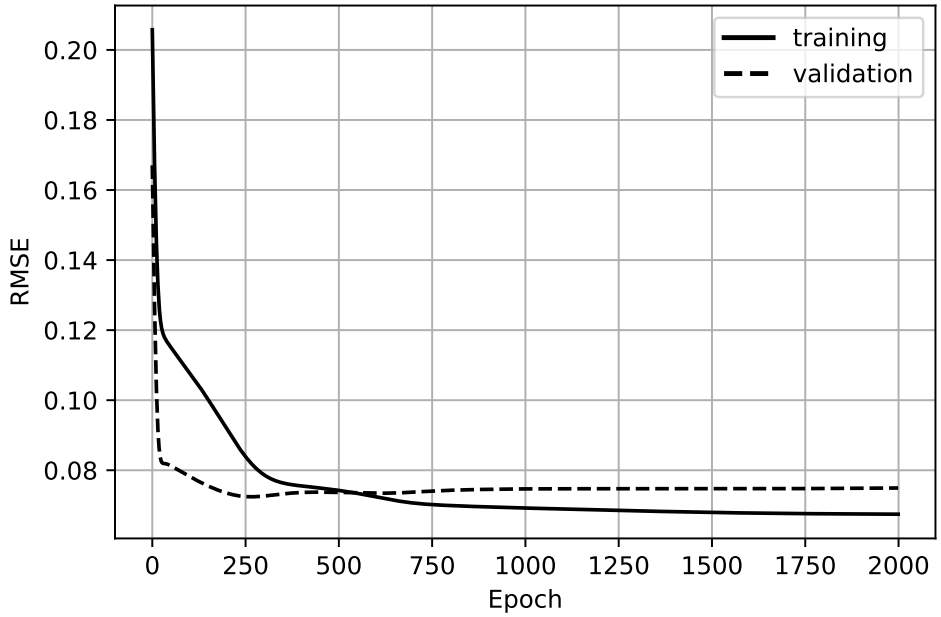

Figure 3: History of training and validation RMSE

Once built the LSTM neural network, the three models can now be compared. Since out-of-sample forecasts are based on weights or parameters learned from data in the training and validation set, we will henceforth refer to these two sets as just the "training set".

Starting from the GARCH(1,1) and EWMA models, their respective parameter values are estimated through MLE and shown in Table 3.

	$\boldsymbol{\omega}$	$\boldsymbol{\alpha}$	$\boldsymbol{\beta}$
GARCH	0.0000543	0.1246538	0.8622846
EWMA		0.1063281	0.8936719

Table 3: Estimated parameters for GARCH(1,1) and EWMA

The log-likelihood function is maximized through the Nelder-Mead method[6], or simplex method. The results for EWMA are expressed making use of the equivalence relations for which $\beta = \lambda$ and $\alpha = 1 - \lambda$.

[6] For further details, see Gao and Han (2012)

Notice that in the EWMA, the optimal value for λ is largely different from both 0.97 and 0.94, which are the values suggested by RiskMetrics for monthly and daily data respectively.

The long-run volatility, V_L, that is implied by the estimated GARCH(1,1) model parameters can be obtained as follows:

$$V_L = \frac{\omega}{1 - \alpha - \beta} = 22.34\%$$

Notice that this results to be higher than the actual sample mean, equal to 15.63%, thus implying the GARCH to have a slight upward bias in realized volatility.

We report in Table 4 the in-sample and out-of-sample performance measures for the LSTM model as well as the EWMA and GARCH models.

	RMSE		**MAE**	
	In-sample	**Out-of-sample**	**In-sample**	**Out-of-sample**
EWMA	0.0786	0.1059	0.0520	0.0606
GARCH	0.0784	0.1022	0.0543	0.0598
LSTM	0.0696	0.0815	0.0411	0.0453

Table 4: In-sample and out-of-sample performance measures

We find that the LSTM neural network produces better results than EWMA and GARCH(1,1) both in-sample and out-of-sample. In particular, the LSTM improves the RMSE of the EWMA and GARCH models by 20.25% and 23.04% respectively, and the MAE by 24.25% and 25.24% respectively. Moreover, LSTM produces also smaller differences between out-of-sample and in-sample measures. Indeed, whilst the average differences for the econometric models correspond to 0.026 for the RMSE and 0.007 for the MAE, the LSTM yields a difference of 0.012 and 0.004 for the respective measures. That means that apart from the absolute results, the LSTM is relatively better at dealing with the problem of overfitting.

Finally, we leave the reader with a comparison of the out-of-sample forecasts among the three different models (see Figure 4).

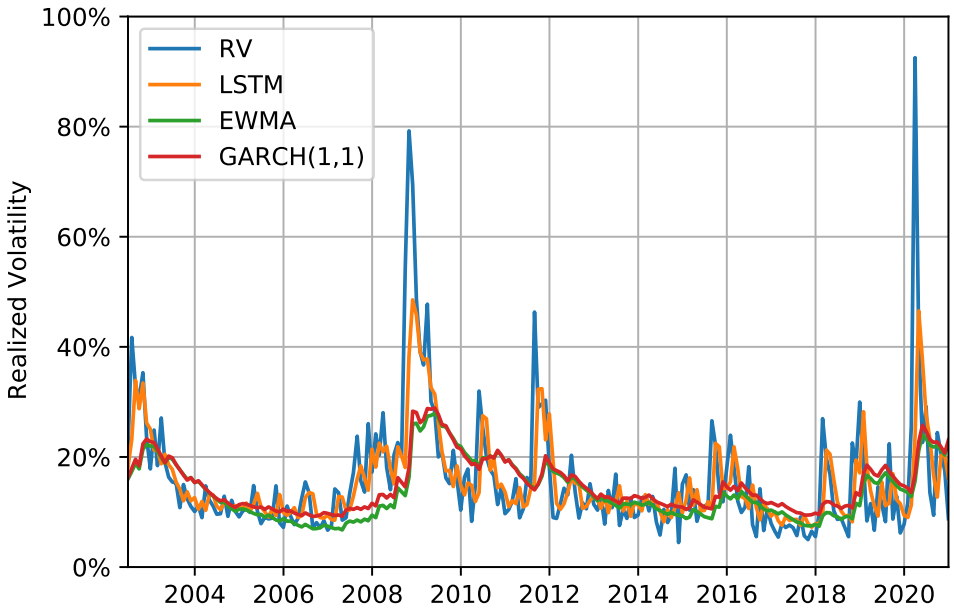

Figure 4: Compared out-of-sample forecasts