University of Milan Bicocca

**School of Science**

**Department of Computer Science, Systems and Communication**

**Bachelor of Science in Computer Science**

# Volatility prediction with Recurrent Neural Networks for options trading

**Speaker**: *Antonio Candelieri*

**Co-Rapporteur**: *Silvio Maria Enrico Bencini*

**Final test report of:**
*Luca Poli*

*Serial no. 852027*

**Academic Year 2021-2022**

# Index

# Introduction

Machine Learning is being adopted in an increasing number of application domains; one in which it has found particular success is undoubtedly finance, and specifically financial markets. Here it has been applied in a variety of ways, from portfolio management to high-frequency trading, with the use of increasingly innovative models and techniques; especially in recent decades, where computational resources and data are no longer a major obstacle. Neural Networks, in particular, have received renewed interest; thanks to the reduction of the aforementioned problems, it is indeed possible to exploit their power and versatility. In addition, more and more variants are being developed; such as *Recurrent Neural Networks* (*RNNs*), which are specialized to work with sequences of data (such as the time series that make up most financial data).

Thus, the objective of this thesis is to implement a predictive system; based on *Long Short Term Memory* (*LSTM)* networks, a particular class of RNNs, that is able to anticipate the volatility of a stock price. Finally, it is intended to make profit from such prediction, with a short-term trading system based on options. The paper is intended as a practical introduction; that is, it will first address the broad theoretical concepts, and then propose a possible solution.

The first chapter will be divided into three sections: in the first, I will introduce Machine Learning; in the second, I will discuss Neural Networks, moving from the simplest to the most complex ones, with a focus on problems and their solutions, then ending with an in-depth discussion of the Recurrent version; finally, in the last part, I will address the topic of options and volatility.

The second chapter is a general description of the project: first I will describe the data involved and its processing; then I will define the Machine Learning models adopted and compared; then I will outline the strategies adopted by the trading system; finally I will proceed with the presentation of the results.

# Chapter 1: Theoretical concepts

This chapter introduces the theoretical concepts behind the project, both from the ML and Neural Networks side and from the financial side.

## *1.1 Brief background on Machine Learning (ML)*

Following the definition proposed in [1], we define Machine Learning as a set of techniques and algorithms that allow the automated solving of complex problems, which would be difficult to solve with the conventional programming approach. Such a system, in its most general aspects, starts with a specification that defines the problem; that is, in software terms the final task of the program. It then designs and then implements the set of rules that makes up the specific solution. However, this approach is very difficult, in practice almost impossible, to apply in situations where the specification is clear, but the set of rules to be defined is not; this is where the ML comes in, which allows the computer to learn from the specification itself the long, and exhaustive, set of criteria that make up the solution. It solves the problem indirectly: it creates and updates a model based on the input data, learning common rules and patterns; and uses that to produce the solution to the problem.

An example is the detection of textual characters in images. Suppose we have a dataset containing example images, i.e., with the corresponding solution (in jargon called a *label*); the computer scientist should investigate these images, understand their common factors, and finally construct a generic procedure that works on new images. As clear as the specification is, the generic set of rules is complicated to establish. So, applying the tradition approach would be complex, so we can use ML techniques.

To fully understand how ML works, it is good to delve into the problems it solves and the mechanisms of its models. The problems solved, as proposed in [2], fall into several macrocategories:

- *Classification*: assigning, to each element, one or more appearance categories;
- *Clustering*: separation of a large data set into small subsets, the number and characteristics of which are not defined a priori;
- *Regression*: inference of a single true value from a set of information. It differs from classification because the notion of closeness between two values is present, which is not always present between two categories;
- *Ranking*: sorting of items based on certain criteria, for example, sorting the websites most relevant to a search;
- *Dimensionality reduction*: transforming the representation of an element, into a smaller representation.

*(Note that this separation is purely formal, and areas often collapse into broader or narrower categories.)*

Each ML model is characterized by two categories of internal variables:

- Hyperparameters, the quantities and roles of which vary according to type; correspond to model settings and therefore govern all the internal dynamics of the model; are established a priori by the designer.
- Parameters, on the contrary, change during the life time of the model, depending on the hyperparameters are defined used and changed.

The models, before they can solve the problem, must be trained; that is, they must learn the patterns and rules, continuously varying their parameters; in order to solve the problem with some reliability. This state is referred to as the training phase, precedes (sometimes coincides with) the operation phase itself, and can take a large amount of data, time and computational resources. Depending on how it occurs, we distinguish different types of learning models; as explored in more detail in [1], the most widely used are:

- *Supervised Learning*: models in this category are trained with labeled (jargon: *labelled*) data, that is, data whose meaning and end result we know. They exploit this knowledge and learn by trial and error, in which they vary their internal parameters based on the deviation between the obtained value and the known value. They usually tackle classification and regression problems.
- *Unsupervised Learning*: in this category, models usually solve clustering problems, and are trained with unlabeled data; therefore, the processor will have to recognize similarities, common features and correlations independently from the initial data; in order to classify them into different categories.
- *Semi-supervised Learning*: is a middle ground between the previous two, models in this category learn through partially labeled data; use clustering techniques to recognize subgroups, and classify such by exploiting Supervised learning techniques with labeled data.
- *Reinforcement Learning*: in this case there is no real training phase and the model is not aware of the final result (so the data is not labeled), the model trains, during operation, based on a score (score) that is given to each computation; with the goal of maximizing it. So, as time passes, the model will continue to update and adapt to new situations, without the need for retraining.

## *1.2 Artificial Neural Networks.*

Artificial Neural Networks (ANNs) are an ML model that is inspired by the working principles of the human brain. Just as it is composed of cells called neurons interconnected by synapses; Artificial Neural Networks are composed of interconnected layers of neurons. Computation starts from input neurons, evolves from layer to layer, and is propagated via weights (which are the parameters of the model) to output neurons.

Each neuron receives input data from outside or previous neurons, and computes through a function, called Activation (or *Activation*), an output data. So, the multi-neuron structure that characterizes the network allows us to combine different functions and build a more "intelligent" system than other ML models. It, however, is more complex to train and therefore requires more data and computational power. Due to the lack of these resources ANNs, despite being born as early as the 1960s (in '58 Rosenblatt introduced the first single-neuron network: The Perceptron) will only be studied and used extensively since the early 2000s.

ANNs are, typically, a Supervised Learning model; in fact, during the training phase the weights are modified based on an error function (called *Loss Function*) that expresses the distance between the predicted and the given values. Typically it is used to solve classification problems; but, especially for some variants, it also applies well to prediction problems.

For example, a solvable problem might be the classification of an image; the input data are the pixels of the image while the output is the class to which the image should belong; during training, the relevant pair (input, output) of data is passed to the network and the internal link weights are changed due to the computation of the error function, which expresses the accuracy in classification.

### 1.2.1 The basic principles of neural networks

This section will introduce the operating principles of artificial neural networks, focusing mainly on the fundamentals of standard architectures. Specifically, the most basic ANN will be discussed first: the Perceptron and its operating components; then moving on to multi-layer ANNs, and finally investigating the training and overfitting problem.

#### 1.2.1.1 The Single Neuron: The Perceptron

The simplest neural network consists of a dummy layer of input neurons and a single neuron, called Perceptron, that performs the calculations and returns the output.

The training dataset, defined as dataset (*D)*, is divided into subsets of equal size. called *batches* that include multiple instances taken randomly. At each cycle, called: epoch (epoch) the network is trained by taking input batches in random order. Each instance is the input and output pair in the form of $(\overline{X}, y) \in D$; where $\overline{X} = [x_1, \ldots.., x_d]$ contains *d* variables representing the input of the network, $y$ contains the value of the output variable that has been previously observed (and therefore real).

The input layer is dummy, and therefore often uncounted, because it does not perform calculations, and merely propagates input values into the network. It consists of *d* nodes that propagate input variables $\overline{X} = [x_1, \ldots.., x_d]$ via weights $\overline{W} = [w_1, \ldots.., w_d]$ to the output node. It performs two calculations: first the weighted sum of the inputs $[\overline{\mathrm{W}} * \overline{\mathrm{X}}]$; and then, it applies the above activation function to the result. $\Phi$. The final product is:

$$\hat{y} = \Phi(\overline{\mathrm{W}} * \overline{\mathrm{X}}) = \Phi(\sum_{\mathrm{i}=1}^{d} w_i \, x_i)$$

During the training phase, at each epoch, the error of the prediction is used $E(\overline{X}) = y - \hat{y}$ in the form of a loss function *L*. It varies depending on the activation functions, the type of problem, and the situation; it is used to best express the "distance" between the true value (y) and the calculated value ($\hat{y}$). The optimization, i.e., finding the minimum, of this function allows us to obtain the update values used to change the weights $\overline{W}$; these values are adjusted by a hyperparameter called the *learning rate* (α), which amplifies or reduces them. Algorithms and mathematical approaches are used to reach that point of optimum, which usually make use of the gradient.

We can consider as an example the Perceptron formulated by Rosenblatt in 1958. In this case the training instances are in the form $(\overline{X}, y)$ with $\overline{X} = [x_1, \ldots.., x_d]$ e $y \in \{-1,1\}$ which is a binary variable indicating class membership; thus, the activation function is the sign function: sign(x) -> {+1, -1} depending on the sign of x; while the loss function is quadratic:

$$Minimaze_{\overline{W}}\, L = \sum_{(\overline{X},y)\in D} (y - \hat{y})^2 = \sum_{(\overline{X},y)\in D} (y - sign\{\overline{W} * \overline{X}\})^2$$

Originally Rosenblatt did not have the mathematical tools available to find the optimum and optimize the weights, therefore, he used systems implemented directly in hardware. In contrast we, can use an approach defined as *Gradient Descend*, in which the gradient of the loss function is calculated in order to update the weights. Specifically, the value of the gradient is multiplied by the *learning rate*, and finally subtracted from the weights. However, this method requires a differentiable function; however, the sign function does not meet this requirement, and the exact calculation of the gradient is not possible; therefore, as proposed in [3], an approximation of the loss function called smooth is used:

$$\nabla L_{smoo} = \sum_{(\overline{X},y)\in D} (y - \hat{y})\overline{X}$$

After each instance of training $(\overline{X}, y)$ the weights $\overline{W}$ are updated as follows:

$$\overline{W} \leftarrow \overline{W} + \alpha(y - \hat{y})\overline{X}$$

*(where α is the learning rate)*

### *1.2.1.2 The activation functions*

They are a hyperparameter of ANN, so they are established a priori, and each neuron can have a different one.

There are several types, and their choice is critical during ANN design. It varies depending on the situation and the type of problem; for example, if we need to predict a real value we would use the Identity function, if we need to perform a binary classification (as in the introductory example) we would choose a sigmoid, with multiple classes we would opt for a softmax. However, the evaluation of the most suitable function also depends on the complexity of such, especially if it will later be combined with others. In fact, as further discussed in [3], some functions are simpler to derive, and the calculation of the gradient is more "linear"; therefore, finding the optimum of the loss function is faster.

Some activation functions are:

- Identity: $\Phi(x) = x$
- Sign: $\Phi(x) = sign(x)$
- Sigmoid: $\Phi(x) = 1/1 + e^{-x}$
- Tanh: $\Phi(x) = (e^{2x} - 1)/(e^{2x} + 1)$
- ReLu: $\Phi(x) = \max\{x, 0\}$
- Hard Tanh: $\Phi(x) = \max\{\min\{x, 1\}, -1\}$

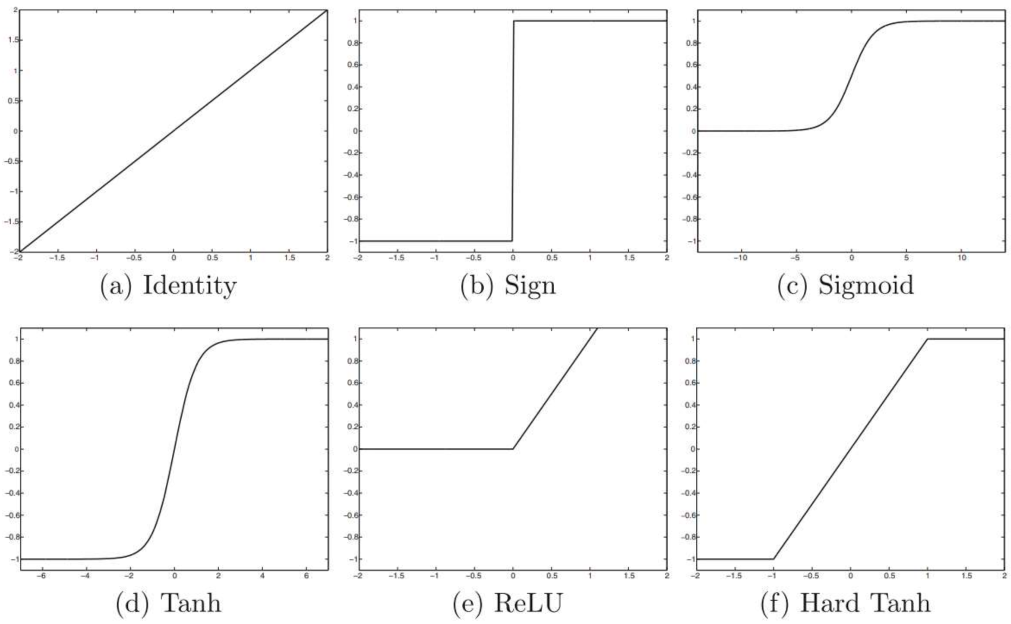


*Figure 1: Some activation functions*[1]

We note, both from the formulas, and from the graphs in **Error! Reference source not found.** That they are very different from each other. Sigmoid and Tanh are used for classification problems; Identity for prediction problems; finally, ReLu and Hard Tanh are similar versions of Sigmoid and Tanh, but they are easier to derive (given the more "pointed" forms) and therefore quicker to optimize.

[1] Neural Networks and Deep Learning: A Textbook, Charu C. Aggarwal, p. 13

In the case of the Perceptron we have only one to choose from, however the real learning power (i.e., "intelligence" of the model), is achieved by combining many of them together. This topic is addressed later.

### *1.2.1.3 Loss Function in Detail*

It too is a hyperparameter, is common throughout the network, and the choice comes from the type of problem and the goal of the ANN; because it is necessary to make sure that the output of the network is sensitive and sensible with respect to the application in question.

For example, it would be unreasonable to use a function such as the root mean square deviation for a classification problem; this is because the concept of closeness between values is present in that function, which is not present among class labels (example: it makes sense to say that 3 is close to 4, it does not make sense to say that dog is close to cat).

There are several, and they are often derived from other ML models, two for example are:

- Mean Squared Error Loss:

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \widehat{y_i})^2$$

With n = number of instances in D

Used for regression problems, it is a positive value that indicates the deviation between predicted and actual values.

- Cross-Entropy Loss:

$$L = \sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Where:

- *M*: is the number of classes;
- o: is the observation in question
- $y$: is a binary indicator (0,1) that indicates whether *c* is the correct classification for o

- $p$: is the observed probability that o is of class *c*
- The formula corresponds to a single instance

Used for classification problems, the smaller the value, the more accurate the model will be.

The formulas shown correspond to the Loss function evaluation in the Perceptron case; in fact, as analyzed by [3] , in the case of multilayer networks the calculation is much more complex.

#### *1.2.1.4 Multilayer Neural Network*

The Perceptron has only one computational layer, the output layer, and one input layer, the input layer; in contrast, Multilayer (multilayer) networks include between the above two additional internal layers, called *hidden* because they do not interact with the external interface. The latter perform the main computations, plus they can be of various sizes and use different activation functions. The choice of the number of layers, the nodes in each and their functions can result in very different networks; mainly such hyperparameters influence the "intelligence" of the model, the difficulty of the training phase and the real effectiveness of the latter.

In addition, the connections between layers and neurons can also change and give rise to very different architectures, there are all kinds, some of the most important of which are listed next.

- Fully connected network: particular ANN where each neuron is fully connected with all other elements in the network; it is not widely used in practice and is mentioned only from a theoretical point of view.

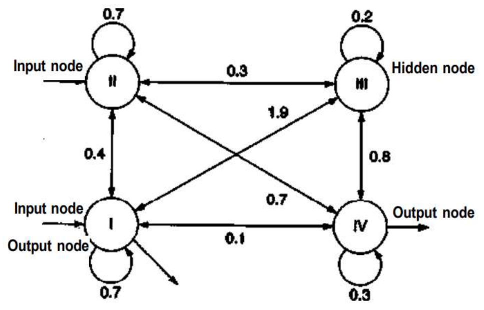


*Figure 2[2] : Typical Fully Connected network*

- Feed-Forward network: considered as the conventional architecture, each node in an inner layer receives input from all nodes in the previous layer and propagates the result to all nodes in the next layer; starting from the first input layer to the last output layer.

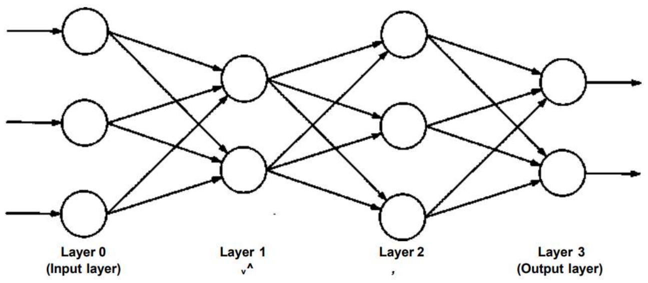


*Figure 3[3] : Typical Feed Forward Network*

- Convolutional neural network: specific architecture created to work with grid-structured data, which also have correlations and dependencies between spatially close elements. An example is an image; in fact, in addition to having two dimensions of height and width, we can add a third, the color of each pixel, which is precisely spatially dependent on the other 2. To process such data, the input layer of this architecture therefore presents a three-dimensional structure.

[2] Elements of artificial neural networks, M. kishan, C.K. Mohan, S. Ranka, p. 17
[3] Elements of artificial neural networks, M. kishan, C.K. Mohan, S. Ranka, p. 20

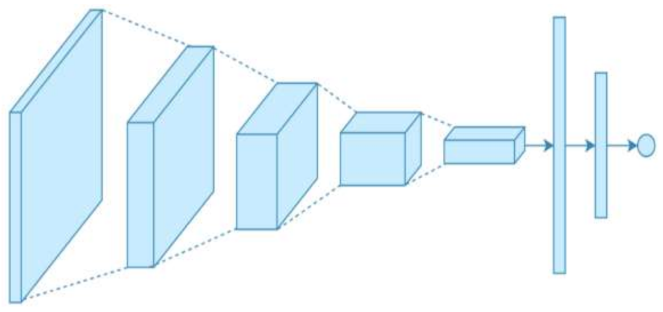


*Figure 4: Typical Convolutional Network*

- Recurrent neural network: used to work with sequences of data (such as time series and sentences), through the use of loops in the network, allows learning the correlations between neighboring elements in a sequence.

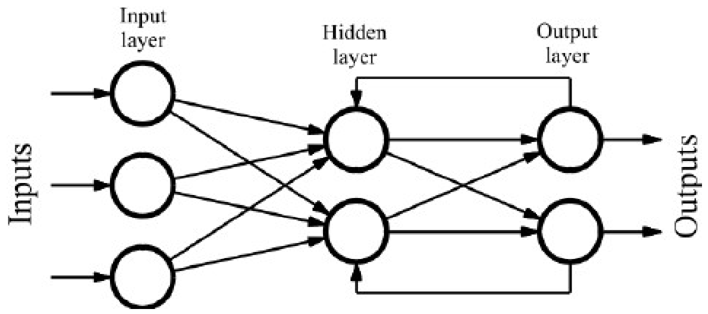


*Figure 5[4] : Typical Recurrent Network*

Single-layer architectures are very limited, in contrast multi-layer ones are not. The real expressive power, to which the growing popularity of ANNs is due, lies precisely in the possibility of combining, via different layers, multiple nonlinear activation functions; this leads to a model that is relatively easier to train and much more expressive, allowing any function to be approximated, theoretically. However, the use of nonlinear functions is necessary; because it is mathematically provable (as done in [3] e [4]), that the combination of several linear functions (such as Identity) can be approximated by the use of only one, and thus brings no benefit.

[4] Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Scheme-of-a-recurrent-neural-network_fig2_347917707

### *1.2.2 Feed-Forward Neural Network (FNN)*

In this section I will introduce the characteristics of the conventional Multilayer network, elaborating on the positive aspects and investigating the complexities involved in this architecture compared to the Perceptron. I will also analyze the training phase in detail.

#### *1.2.2.1 The architecture in detail and its advantages*

As mentioned earlier, the Feed-Forward type architecture represents the conventional structure of a Multilayer ANN. The Perceptron can be seen as its simplified version, so the features are in common; however, the learning capability of the Perceptron is reduced and presents the difficulties of the FNN in a minimal way.

The structure of the FNN is composed of 3 types of neuron layers:

- Input layer: present in a single quantity, it is the only one into which external values enter; it does not perform computation, the number of component nodes is equal to the features input to the network;
- Output layer: present in a single quantity, it is the only one where values come out externally, the number of component nodes is equal to the output features;
- Intermediate or hidden layer: placed in variable amounts between the previous two layers, the number of nodes is also variable.

Each node in an inner layer is interconnected with all nodes in the previous and next layers, the computation starts from the input layer and ends in the output layer going from layer to layer without jumps or cycles.

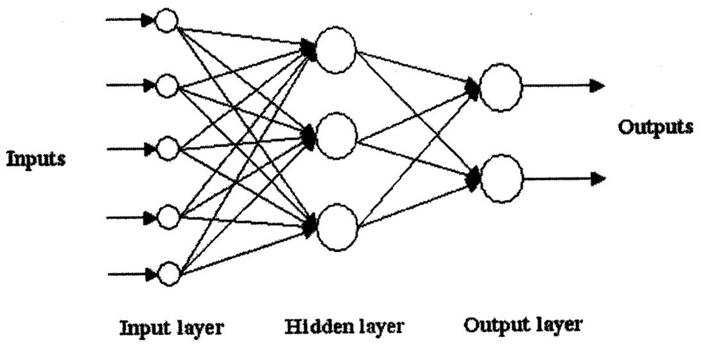


*Figure 6: The layers of an FNN and the order of computation.*[5]

Such an FNN can be viewed as a computational graph that combines activation functions by mathematical composition; for example, taken an *m* layer, with *k* nodes, that evaluates the function $g(\cdot)$ and an *m+1* layer that applies the function $f(\cdot)$; thus, each node present in *m+1* computes, with different weights, the function: $f(g_1(\cdot), \dots\ g_k(\cdot))$. If such composition does not use only linear functions, it becomes a very powerful tool; because it allows solving nonlinearly separable problems; i.e., with a complex and arbitrary decision surface (as deepened in [5]). Thus, it behaves as a universal function approximator, i.e., it allows any function to be computed theoretically. This theoretical limit is achievable only by a network with sufficient neurons and trained perfectly; however, the more the number of nodes in a layer increases the more data is required for the training to be effective.

#### *1.2.2.2 The Training Phase*

In the case of the Perceptron this phase was solved with the Gradient descend algorithm, which was simple to compute given the use of only one function. In the case of a multilayer network, the functions applied in the last layers are a very complex composition, which also depends on the weights in the previous layers; therefore, the calculation of the gradient is very difficult. To solve this problem, the *Backpropagation* algorithm is used, which calculates the gradient by going backwards. However, it exposes the network

[5] A brief review of feed-forward neural networks, M. H. Sazli, p. 13

to another problem, called *Vanishing and Exploding Gradient*, which makes the training unstable. These concepts are discussed in more detail in the following subsections.

#### *1.2.2.2.1 The Backpropagation Algorithm*

The invention of backpropagation was a real turning point in the history of ANNs; in fact, its conception helped greatly to popularize them over the past two decades.

This algorithm bases its operation on the chain rule of differential calculus; that is, it calculates the final gradient in terms of the sums of the products of the local gradients, calculated on all paths starting at the node in question and ending at the output node. This calculation, which involves an exponential series of paths, is made possible by applying the principles of dynamic programming.

Specifically, it involves two phases: Forward and Backward. The first is the traditional phase of supervised training; data is given as input to the network, and this results in a chain computation that produces the final result; finally, it is compared with the known value and the error has to be computed. The second stage is responsible for this computation, which is accomplished by exploiting the chain rule of differential computation; a simplified explanation follows.

We define $e_0$ as the error function detected at the generic output node *o*. The chain rule distributes the gradient of $e_0$, on each node in the previous layer to which it is connected *o*, depending on the weight $w_i$ of the i-th link; then each node redistributes, in the same way, the received gradient to the previous layer; and so on until the input layer is reached. Put another way: each internal node *h* receives from each output node *o*, a weight modification factor equal to the gradient of $e_0$, proportional to the weights of the connections connecting *h* with *o*.

Taking an example, where we have a node in a first layer with only one weight *w*, two nodes in the second layer, and an output node:

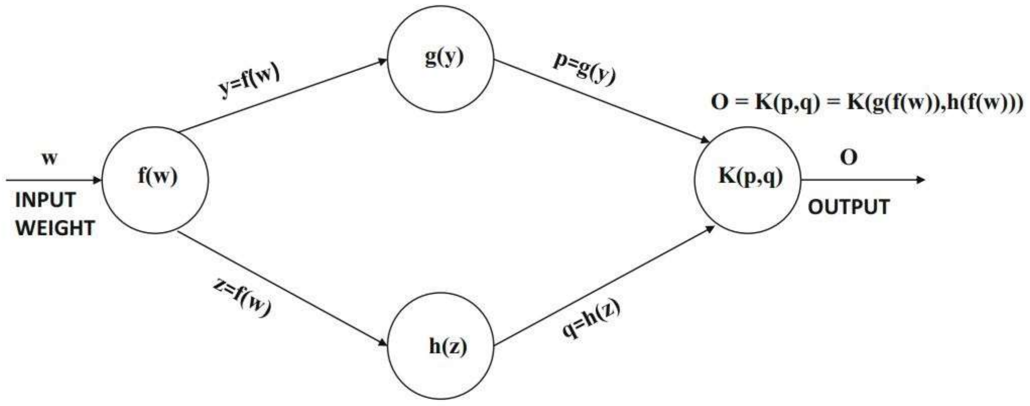


*Figure 7[6] : Example graph*

So, in the backward phase, the output node distributes, according to the weights, the error to the two previous nodes, which will do the same thing with the node of the first layer. The comprehensive explanation is deepened in [3]; in summary:

$$\frac{\partial o}{\partial w} = \frac{\partial o}{\partial p} * \frac{\partial p}{\partial w} + \frac{\partial o}{\partial q} * \frac{\partial q}{\partial w} =$$

$$\frac{\partial o}{\partial p} * \frac{\partial p}{\partial y} * \frac{\partial y}{\partial w} + \frac{\partial o}{\partial q} * \frac{\partial q}{\partial z} * \frac{\partial z}{\partial w} =$$

$$\frac{\partial K(p,q)}{\partial p} * g'(y) * f'(w) + \frac{\partial K(p,q)}{\partial q} * h'(z) * f'(w)$$

*1.2.2.2.2 Vanishing and Exploding gradient*

The backpropagation algorithm, despite its effectiveness, exposes the network to some problems. The first of them, although more negligible are local minima; in fact, when the surface of the loss function is very irregular, there is a risk that the algorithm will get stuck in a local minimum. The second, and most important problem, defined as Vanishing and Exploding

[6] Neural Networks and Deep Learning: A Textbook, C. Aggarwal, p. 22.

gradient; it can lead to instability during training and cause a real paralysis of the network.

Specifically, this problem, is caused by the exponential complexity of the algorithm; where slightly irregular changes to the weights of the first few layers, propagate in an avalanche to the rest of the network; leading the gradient to take on values too large (Exploding) or too small (Vanishing); and thus the algorithm will not achieve convergence, because the increments of the weights are too large to be accurate, or too small to be effective.

This problem is affected by the number of layers; in fact, the more layers, the greater the training complexity. One strategy to reduce it is to use some activation functions over others, such as: ReLu and Hard Tanh; because they have derivatives that are simpler to compute, and thus make the composition less complex. Another method is based on the dynamic use of the learning rate.

#### *1.2.2.2.3 Learning Rate*

The learning rate (α) is a very important hyperparameter. It takes part in the training phase, in which it is multiplied by the gradient; in order to obtain the final factor by which to change the weights. So, by varying from 0 to 1, it acts as a learning regulator, that is, it reduces or does not reduce the impact that each epoch has on the network.

This hyperparameter (as explored in detail in [6]) is never fixed but, for better and more consistent training, changes dynamically. There are two main approaches.

- *Decay based*: in which the learning rate decreases from epoch to epoch, depending on a rate regulator *k*, which controls the decay. It is usually implemented in two alternatives:
  - *Exponential* decay: $\alpha_t = \alpha_0 * e^{-k*t}$ (where *t* is the current epoch number)
  - *Inverse* decay: $\alpha_t = \alpha_0/(1 + k * t)$ (where *t* is the current epoch number)

- *Momentum based*: is a more complex technique, which adjusts the learning rate according to momentum; that is, modifying it based on the average of the last movements. This effect is achieved in the following way:

$$\overline{W} \Leftarrow \overline{W} + \overline{V}$$

$$\overline{V} \Leftarrow \beta * \overline{V} - \alpha * \frac{\partial L}{\partial \overline{W}}$$ (with β hyperparameter momentum controller)

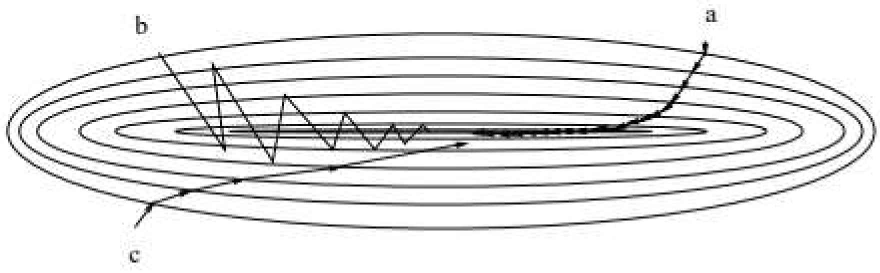


*Figure 8[7] : The gradient descend in parameter space. a)with a low learning rate. b)with a high learning rate. C)with a high learning rate with momentum*

Other strategies, to make the training phase more consistent, rely on the use of learning rates that vary for each parameter. Because parameters with more complex partial derivatives, they tend to oscillate more, and need a more dynamic learning rate. The most important ones, briefly seen, are:

- AdaGrad: emphasizes consistent movements, penalizing partial derivatives that tend to have large fluctuations. By assigning the value $A_i$, aggregate of all partial derivatives, of the *i-th* parameter an update thus defined:

$$A_i \Leftarrow A_i + \left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i$$

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i$$

[7] An introduction to neural networks, B. Krose, P. van der Smagt, p. 37.

- RMSProp: in contrast to AdaGrad, where the aggregation of $A_i$ led to premature slowing down; instead of summing, an exponential average adjusted by a decay factor is used $\rho \in (0,1)$:

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i$$

$$w_i \Leftarrow w_i - \frac{\alpha}{\sqrt{A_i}}\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i$$

- AdaDelta: similar to RMSProp, but eliminates the need for the global learning rate hyperparameter $\alpha$ by calculating it from previous updates:

$$w_i \Leftarrow w_i - \sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i$$

$$\delta_i \Leftarrow \rho \delta_i + (1-\rho)\left(\sqrt{\frac{\delta_i}{A_i}}\left(\frac{\partial L}{\partial w_i}\right)\right)^2 \quad \forall i$$

- Adam: corrects the problematic initialization bias of RMSProp, in which early updates affected the entire learning phase. It then uses: an exponentially smoothed version $F_i$ of the gradient; a parameter $\rho_f$ similar to $\rho$ but incorporating momentum; and the learning rate $\alpha$ is replaced with $\alpha_t$ which depends on the iteration index *t*:

$$A_i \Leftarrow \rho A_i + (1-\rho)\left(\frac{\partial L}{\partial w_i}\right)^2 \quad \forall i$$

$$F_i \Leftarrow \rho_f F_i + (1-\rho_f)\left(\frac{\partial L}{\partial w_i}\right) \quad \forall i$$

$$w_i \Leftarrow w_i - \frac{\alpha_t}{\sqrt{A_i}} F_i \quad \forall i$$

$$\alpha_t = \alpha\,(\frac{\sqrt{1-\rho^t}}{1-p_f^t})$$

### *1.2.2.3 Overfitting*

This is an existing problem in all ML models, but it is well present in ANNs (and FNNs) because of the large number of parameters possessed by a complex network. It represents the loss of the generalization ability of the model.

To solve this problem, after the Training phase, there follow two phases[8] of verification: *validation* and *testing*; in which the model will deal with data not yet used (called *validation set* and *test set,* respectively). In the former, the results produced will be used to track, and improve the generalization capability of the network; in the latter, they will be used to have a final evaluation of the model.

#### *1.2.2.3.1 Cause*

It occurs when the performance on the training data is much higher, compared to that obtained with data never seen by the network. This happens because the FNN has become too specific to that data; therefore, it is as if it has memorized it, rather than learning the logical patterns among it.

The general cause of this phenomenon is due to three factors: few training data; or poor representation, provided by them, of the set; too complex network. They are related, because a network that is too complex, that is, with too many parameters, needs more data, and, if they are not representative of the generic set, then they are of little use.

Data scarcity was one of the main reasons, why ANNs were not used until the 2000s. To date, the problem of overfitting, is much discussed and addressed with different techniques.

#### *1.2.2.3.2 Solutions*

[8] *What are artificial neural networks?*, A. Krogh, p. 197

The simplest solution would be to increase the number of data; however, when they are limited we can resort to more or less complex systems. Some of them are introduced below.

- Hyperparameter tuning: a very common practice, in which you test the network by varying the hyperparameters and measuring the error on the validation set; finally, you use the set of hyperparameters that makes the network perform better.
- Regularization: a regularization hyperparameter is introduced that acts as a penalty; it goes to reduce the value that can be taken by the parameters; and thus, the individual influence that each of them has on the result produced. In fact, it will be similar to having a NN with fewer parameters, which turns out, however, to be more "intelligent."
- Early stopping: a very common strategy in which, during training, the error is also tracked on the validation set. When the error produced, on the two data sets, diverges too much, early stopping is done.
- Parameter sharing: a practice that can be adopted in situations where correlation between two or more functions computed in multiple nodes of the network is useful. In these cases, there is a tendency to use specific architectures, such as RNN and CNN, that exploit this very principle.
- Trading Off Breadth for Depth: you decrease the number of nodes per layer, in favor of more layers. This reduces the number of connections (and thus parameters) without making the model less "intelligent"; although the training phase becomes more difficult (due to convergence problems).
- Unsupervised pre-training: it precedes the training itself, with a phase involving each layer individually. In it, pre-training, that is, node weights are initialized, through an unsupervised algorithm.
- Ensemble Methods : family of methods common to other ML models (although some are specific to ANNs). They create several basic

models, through a principle that varies according to the method, and then combine them to obtain the optimal one.

### *1.2.3 Recurrent Neural Networks (RNNs)*

RNNs are an unconventional architecture of multilayer ANNs; they are extremely useful in certain applications, but they add additional problems and complexity. I will first address these features, and then introduce a specific type of RNN: LSTM, which is particularly efficient in solving certain problems.

#### *1.2.3.1 The basic principles*

FNNs are designed to work with multidimensional data, where each attribute is independent of the others; thus, they sin in recognizing and exploiting dependencies among them. In contrast, RNNs are designed to work with data that have sequential dependencies, such as time series or text; where one element in the sequence is closely related to the previous or next. They, unlike FNNs, not only recognize dependencies, but also the order of the elements.

In addition, the particular recursive structure of the RNN, allows the use of sequences with undefined length. This would not be feasible with an FNN; because, since each neuron is associated with an element of the sequence, it would not be possible to have a network with an undefined number of neurons.

Finally, RNNs are called *Turing Complete*, that is, they can simulate any algorithm, provided they have enough data and computational power. However, this feature is little used and explored, except with particular variants, because the resources required make it impractical.

All these advantages come from a much more complex structure, which will be difficult to train and more prone to the Vanishing and Exploding gradient problem.

In this section I will introduce the basic features of the architecture and the complexities it introduces.

### *1.2.3.1.1 Architecture in detail*

The architecture of an RNN can be represented by a form called *Time Layering*; in which the network consists of a variable number of layers, and where each corresponds to a position in the sequence (called a *timestamp*). Each layer shares the same set of weights, and has an input, an output, and an internal state that interacts with that of the next layer. In fact, however, the network will consist of a single layer repeated over time, and propagation of the internal state occurs via a self-loop (a recursive loop).

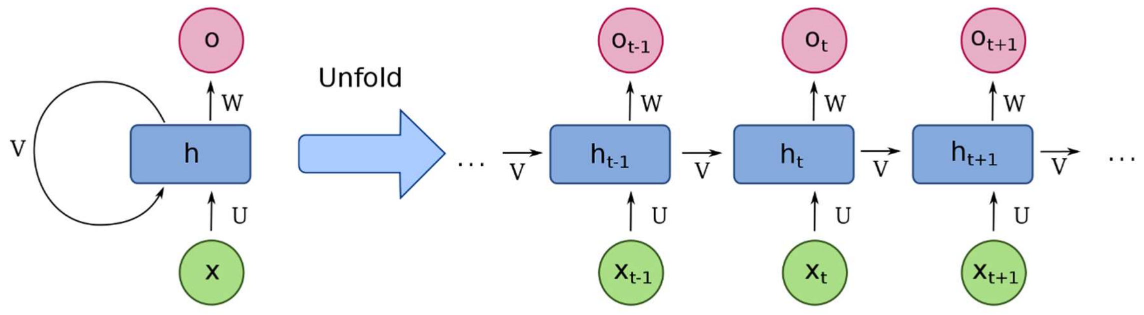


*Figure 9: The real recurrent network and its representation Time Layering*

Thus, at each timestamp *t*:

- $\overline{h}_t = \Phi(U * \overline{x}_t + V * \overline{h}_{t-1})$
- $\overline{o}_t = W * \overline{h}_t$

Where *x*, *h,* and *o* are the 3 layers (input, inner, and output); and *U*, *V,* and *W* the matrices of the link weights connecting the layers. We thus note that the Time Layering form is like an FNN representing what happens to the RNN in each timestamp.

The proposed RNN consists of the most generic case (*many to many*), in which at each timestamp the network has a new input and a new output. In fact, in some situations these components are missing; that is, the network provides only one output at the end of the sequence, or takes only one input at the beginning of the sequence, or all cases in between.

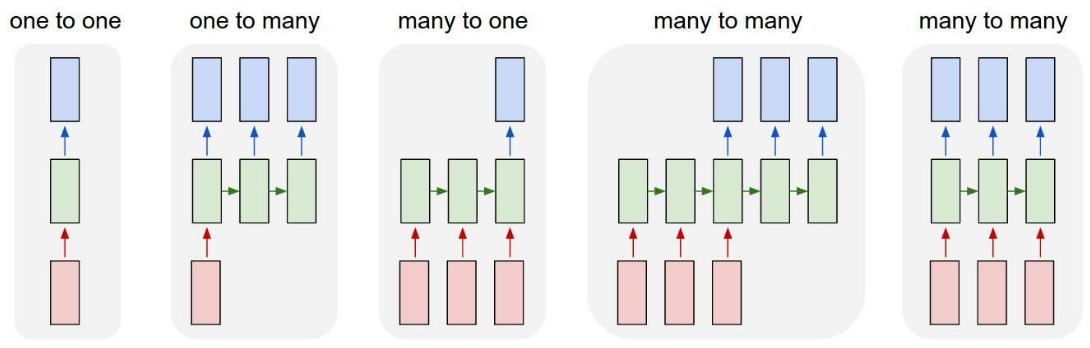


*Figure 10: Different RNNs with input and output layer variables.*

The architecture analyzed so far consists of a single recurrent layer; it is of course possible to have one with multiple recurrent layers in a row, thus increasing the power of the model and the associated training and generalization complications.

There are also several variants, a very important one being the bidirectional one (*Bidirectional*), where each node in the recurrent layer possesses two of its own internal states; one is the traditional one, computed from the previous timestamp, while the other is computed from the next timestamp.

### *1.2.3.1.2 Training*

The training phase for an RNN is more complicated because backpropagation cannot be applied directly, because the algorithm assumes that the weights are always distinct (instead they are shared across timestamps); therefore, the *backpropagation through time* (BPTT) variant is used, which is explored in depth in [4].

The operation of BPTT can be divided into three stages:

1. Similar to the forward phase, the sequence of data in temporal order is provided as input to the network, calculating the error at each timestamp.

2. Similar to the backward step, we calculate the error gradient of each weight by going backward; however, assuming that the parameters in the different time layers are different. To do this, we introduce time variables that should be different but are in fact the same; for example, the weights *W* become different variables $W^{(t)}$ for each timestamp *t*.
3. The gradients of the time variables of the shared parameters are added together:

$$\frac{\partial L}{\partial U} = \sum_{t=1}^{T} \frac{\partial L}{\partial U^{(t)}}$$

$$\frac{\partial L}{\partial V} = \sum_{t=1}^{T} \frac{\partial L}{\partial V^{(t)}}$$

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L}{\partial W^{(t)}}$$

This architecture, as the sequence lengthens, amplifies the vanishing and exploding gradient problem, bringing incredible instability to the training. Because, in addition to having a longer and more complex network; the BPTT, at each timestamp, multiplies the gradient with the same weight $w_t = w$; so if *w* is very large or small it causes the gradient to explode or vanish.

Several solutions to this problem have arisen: strong regularization of parameters, but leading to less expressiveness; a good initialization point for weights; use of Gradient-descend methods that take advantage of second-order derivatives, but require a lot of computational resources; *gradient clipping* technique, which is useful in the presence of shared parameters because it makes the update at each timestamp similar; use of some variants, such as *LSTM.*

### *1.2.3.2 LSTM*

LSTM stands for *Long-Short Term Memory*, it is a particular architecture of the cell of a recursive layer of an RNN; it allows us to solve, in part, the problem of Vanishing and Exploding Gradient; but more importantly to increase the "memory" of the model, increasing the ability to learn long dependencies between data. The structure of an LSTM cell, therefore, is much more complex than that of a classical RNN; in which we recall there being only one activation function, used to compute internal state and output.

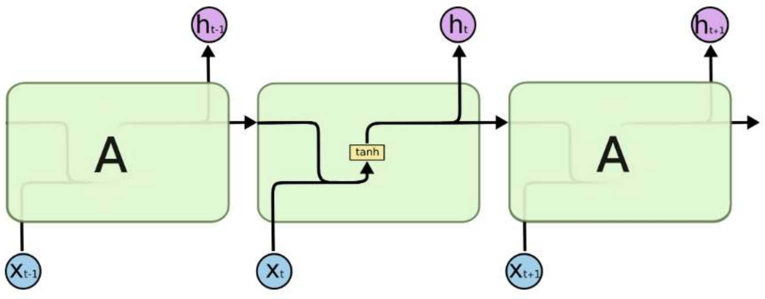


*Figure 11[9] : RNN standard cell structure*

In contrast, the LSTM cell, as thoroughly analyzed in [7] e [8]; is composed of four activation functions, structured to interact in certain ways. The main idea is based on two basic components:

- The internal state: structured like a conveyor belt, it takes its value from the previous timestamp, and is modified during cell computation.
- Gates: a particular operation, consisting of a sigmoid $\sigma$ and a two-factor multiplication, used to adjust the information passing in a line; because, the output of a sigmoid is between 0 and 1. For example, if *x* is the regulating factor, and *y* the line to be adjusted : $y_t = y_{t-1} * \sigma(W_{xy} * x_{t-1})$.

[9] Understanding LSTM Networks, C. Olah

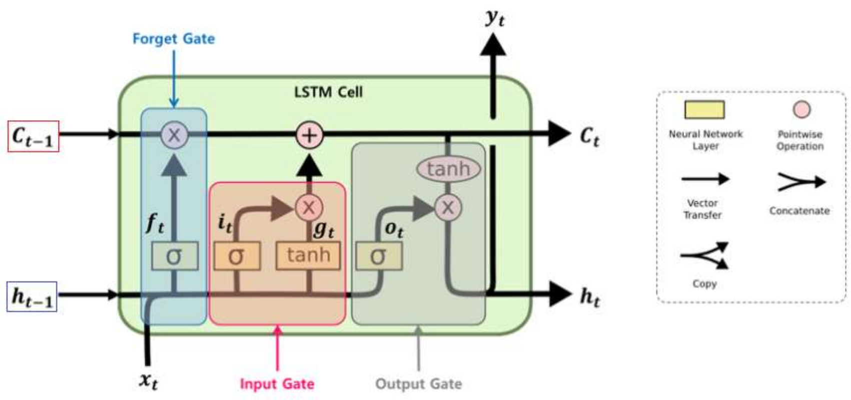


*Figure 12: LSTM cell structure*

Its operation can be divided into three phases, to which the three gates correspond:

- Forget gate: establishes the information to be kept from the previous internal state $c_{t-1}$, from the previous output $h_{t-1}$ and from the input $x_t$:

$$f_t = \sigma(W_{hf} * h_{t-1} + W_{xf} * x_t)$$

- Input gate: establishes the information to be added to the previous internal state $c_{t-1}$, starting from the previous output $h_{t-1}$ and the input $x_t$:

$$i_t = \sigma(W_{hi} * h_{t-1} + W_{xi} * x_t)$$
$$g_t = tanh(W_{hg} * h_{t-1} + W_{xg} * x_t)$$
$$c_t = f_t * c_{t-1} + i_t * g_t$$

- Output gate: establishes the output $y_t, h_t$, starting from the previous output $h_{t-1}$ and the input $x_t$, and the internal state $c_t$:

$$o_t = \sigma(W_{ho} * h_{t-1} + W_{xo} * x_t)$$
$$h_t = y_t = o_t * \tanh(c_t)$$

## *1.3 Options*

Options are a security, that is, an exchangeable, fungible asset with financial value; more specifically, they are derivatives, that is, instruments linked to an underlying on which they "operate." It, in turn, can be any security, although traditionally they are currencies or equities (e.g., stocks); in this thesis, unless otherwise made explicit, we will always refer to options on equities, although the fundamentals do not change with the underlying.

The first options exchanges date back to London in 1690; however, the massive use of the instrument did not develop until the 1970s, thanks to the invention of the Black-Scholes-Merton model, which made it easier to establish the correct price; and the opening of the first exchange, the CBOE (Chicago Board Option Exchange).

### ***1.3.1 The basic principles***

This section will introduce the main concepts; such as the definition, usage, pricing model, how they are combined, and Greeks.

#### *1.3.1.2 What are they?*

An option is a contract entered into, against payment of a premium, between a buyer and a seller (called a writer); it provides the buyer with the right, but not the obligation, to buy or sell a specified underlying asset, at a price and on a certain date. If they provide the right to buy they are called Call, otherwise Put. Traditionally an option controls 100 shares of the underlying asset.

The option is said to be exercised when the right to buy or sell is invoked; The strike price is called the *Strike*, while the date is called the *Expire* (or expiration), because that is when the contract expires. Depending on the difference between the Strike and the current price of the underlying asset, the option has a zero or positive value; different terms are used in this regard:

- OTM (Out of The Money), when the price has not reached the strike (in case of call strike < price, otherwise strike > price);
- ATM (At The Money), when strike and price are similar;
- ITM (In The Money), when the strike has reached and exceeded the price (in case of call strike > price, otherwise strike < price);

For example: *A* enters into a call on an Apple (*AAPL*) with *B*, with a strike at $165 that expires in 3 months, paying a premium of $4; at that time *AAPL* trades at $150, so the option is 10% OTM (i.e., the price must rise 10% to reach the strike). If at expiration *AAPL is* worth less than $165, then *A will* not exercise the option and will lose the premium; if it is worth a little more than $165 (between $165 and $168), *A* will exercise the option by buying *AAPL* at $165 while it is worth more in the market, but counting the premium will still be at a loss; finally, if *AAPL* is worth much more than $165 (>$169) then *A* will have a profit.

One who buys an option (said to be *long*) thus has a potentially unlimited profit and a loss limited to the premium; one who sells (said to be *short*), on the other hand, is in an inverse situation. The investor, depending on his or her view of the market, may decide to:

- Buy a call if it has a very bullish (bullish) outlook;
- Sell a put if it has a neutral or slightly bullish view;
- Sell a call if it has a neutral or slightly bearish view;
- Buy a put, if it has a very bearish view (bearish)

In addition, there are different types of options that vary based on how they are exercised, the main ones being:

- European: allow exercise only on the due date;
- American: which can be exercised even before expiration;
- Bermuda: can only be exercised on certain specific days (in addition to expire)
- Asian: whose final value is given by the average price of the underlying asset over a given period;
- Barrier: can be exercised only if the price of the underlying asset exceeds a predetermined barrier price;

The most commonly used ones (everywhere) are the American ones, but for simplicity, we will always refer to the European ones.

### *1.3.1.3 The value of an option and the Black-Scholes-Merton model*

The intrinsic value of an option is the maximum between 0 and the difference between the price of the underlying and the strike (in the case of a put it is strike minus underlying), and corresponds to the value that the option would have if it were exercised at that time. However, the actual value of an option, which has not yet expired, is greater than the intrinsic value because there is also the so-called *Time* Value; which represents the movement that the underlying can still make in the time remaining to expiration.

There are many components that mainly affect the option premium: the strike, the price of the underlying, the expiration date, the interest rate, the presence of dividends, and the volatility of the underlying. In particular, this last element is very impactful (such as the difference between strike and price and the date), because the higher the volatility (i.e., the standard deviation of the price) the greater the fluctuations of the underlying, and thus the potential profits of long positions (and consequently the premium that buyers will go to pay).

Given the elements involved, the difficult task of assigning the correct price has greatly limited the use of these tools; until the invention of the Black-Scholes-Merton model, abbreviated as Black-Scholes (BS); which solves, at least in part, this problem. It, in fact makes it possible to determine a theoretical value that takes into account the most impactful parameters.

This model, assumes that: the market is random (i.e., not predictable); there are no transaction costs; the interest rate and volatility are constant and known; the returns are normally distributed; the option is European; and there are no dividends over the life of the option. And it stipulates that the price of a call is:

$$C = S * N(d_1) - N(d_2) * \ K * e^{-r*t}$$

$$d_1 = \frac{\ln\frac{S}{K} + \left(r + \frac{\sigma^2}{2} * t\right)}{\sigma * \sqrt{t}}$$

$$d_2 = d_1 - \sigma * \sqrt{t}$$

Where:

- *C*: is the price of the call;
- S: is the price of the underlying asset at that time;
- $K$: is the strike;
- $r$: is the interest accrual;
- $t$: is time to deadline;
- $N$: a normal probability distribution (mean = 0, std. deviation = 1);
- $\sigma$: is the variance of the daily returns of the underlying asset;
- (the formula for the price of a put is similar)

The assumptions previously seen make it unrealistic; but still useful for understanding the main components, and how their variation affects price; of course, as time went on, new models and more "practical" variants (i.e., with fewer assumptions) were invented.

### *1.3.1.5 Greeks*

The term *Greeks* is associated with a family of parameters, specifically Greek letters, that describe the risk exposure of an options position. This exposure can be broken down into several factors, which each Greek describes individually.

They range (for an option controlling 100 shares) from -100 to 100, and indicate how much the change in a given parameter affects the change in price or, in some cases, another parameter; and their definitions are derived directly from the BS model. There are several of them, the 4 mainly discussed, and useful in the thesis, are:

- *Delta* ($\delta$ o $\Delta$): Measures price exposure. That is, it indicates how much the change in the price of the underlying affects that of the option. It is also used as a measure of the probability that the option will escape ITM. For example, one with delta 100 will behave as a long position on the underlying; therefore, a 5% increase in that of 5% leads to a 5% increase on the option as well.

$$\Delta(Call) = N(d_1) \qquad \Delta(Put) = 1 - N(d_1)$$

- *Gamma* ($\gamma$ o $\Gamma$): Measures the exposure of delta to price. That is, it indicates how much the change in the price of the underlying asset affects the change in the option's delta.

$$\Gamma = -\frac{N'(d_1)}{S\sqrt{T}\sigma}$$

- *Theta* ($\theta$ o $\Theta$)*:* Measures the decay caused by time. That is, it indicates how the passage of time (and thus the approach to expire) affects the price of the option.

$$\Theta(\text{Call}) = -\frac{S\sigma * N'(d_1)}{2\sqrt{T}} + r * K * e^{-r} \quad * N(d_2)$$

$$\Theta(\text{Put}) = -\frac{S\sigma * N'(d_1)}{2\sqrt{T}} + r * K * e^{-rT} * N(-d_2)$$

- *Vega* ($v$): Measures exposure to volatility. That is, it indicates how the change in volatility of the underlying asset affects the change in option price.

$$v = S\sqrt{T} * N'(d_1)$$

(For short positions the Greeks will have reverse sign).

### *1.3.1.6 Introduction to Spreads*

*Spreads* are complex combinations of several options, with possibly also the underlying. The component options can vary in type (call or put and long or short), strike or expiration, while keeping the underlying unchanged. Greeks are particularly useful because, being linearly combinable, they help keep track overall of the spread's different exposures.

They are very useful because they allow the investor to expose himself to market factors in ways, which would not be possible to have with the simple underlying. For example, he can cancel out (by setting to 0) one greek to expose himself to the others; or, conversely, he can isolate it by canceling out the others.

One of the simplest is the Bull Call Spread, which involves buying a *C1* call (ITM or OTM) and selling a *C2* call (OTM) with a higher strike and same expiration; in this way the investor still has Bullish exposure to the underlying, caused by C1, but is limited superiorly by C2. So, the maximum profit is the difference of the two strikes (K2 - K1); while the cost of the position, and thus the maximum loss, is the difference between the premium paid for C1 and that obtained from C2 (C1 - C2). Such a spread is ideal when the investor expects a bullish move with a limited extension.

#### *1.3.1.6 General Uses*

Options are widely involved in both short-term trading strategies and long-term investment plans. Precisely because their value is influenced by several factors, which makes them a very versatile, but at the same time very risky instrument.

Large investors (funds, institutions, banks, etc.) usually take advantage of options to *hedge* against risk through *hedging* practices. That is, they buy or sell such instruments, giving up potential profits, in order to offload risks, such as that associated with volatility or large price movements, onto other investors. For example (theoretically) a fund can buy OTM puts, as an insurance against sharp declines.

Instead, smaller investors, or speculators, take advantage of the natural leverage property of these instruments; so that they can speculate more on certain events. In fact, a speculator can get more long or short exposure, at the same cost (relative to the stock); or he can use a spread to expose himself to a specific event, as in the Bull Call Spread.

### 1.3.2 Volatility

Volatility plays a key role in options trading; both in theoretical studies and in the strategies adopted by practitioners in the markets; for these reasons it plays a central role in this thesis and project. It will therefore be explored in more detail in this section.

#### *1.3.2.1 What it is.*

Volatility, in its most generic definition, is the statistic that measures the dispersion of a security's returns over a period. It measures the "magnitude" of a security's movements; therefore, the higher it is, the more large upward or downward movements are expected.

A security is considered statistically safer as long as its returns will be similar to the historical average. Since volatility measures the deviation between average and realized returns, then it is associated with a security's safety index.

For example, taken two security A and B with the same average return, if the volatility of A is greater than that of B; then A will be considered less safe, because the deviation from that return is greater, and therefore the degree of risk is higher.

It is a broad concept that is widely used in different areas of finance; in the context of options, we speak of three of its specializations.

#### *1.3.2.3 Realized (RV) and historical (HV)*

Realized and historical volatility are two similar (often overlapping) concepts, and they differ in the time period considered. If it is in the past, and generally longer (one year), it is called historical volatility; if it is in the future, and generally shorter (one month), it is called future realized volatility. In practice they are calculated using different strategies, the one widely used being *Close to Close*; which defines volatility as the root of the variance of logarithmic returns:

$$x_i = \ln\left(\frac{c_i + d_i}{c_{i-1}}\right)$$

(dove $c_i$ è il prezzo di chiusura del giorno $i$ e $d_i$ il dividendo)

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x}\,)}{N}}$$

(dove $N$ è la lunghezza del periodo e $\overline{x}$ è la media degli $x_i$)

It is usually annualized: $\sigma(annualizzata) = \sigma * \sqrt{252}$

### 1.3.2.3 Implicit Volatility (IV)

It represents the market's expectation of the change in the price of the underling. It cannot be calculated directly, but only approximated by price models (such as Black-Scholes), which owe their inaccuracy to it.

This metric is very complex and caused by several factors; such as: supply and demand for those instruments, macroeconomic condition, expectation of more or less volatile periods, sentiment in general, etc. In addition, each option depending on strike and maturity has a different IV (higher at a distant maturity, and at an OTM strike).

When we speak in general terms of the IV of a market (or stock) we are referring to an aggregation of IVs of the most relevant options; that instrument is called the *IVX volatility index* and represents the general IV of that underlying. It, being an estimate of an expected value, is almost never in line with the corresponding RV, and therefore, there are periods where options for that instrument are relatively overpriced or underpriced.

# Chapter 2: Project

This chapter describes the project in all its aspects, focusing on the motivations behind the design choices. Then in the following sections we will analyze: the data used; how it was processed; the neural network employed and related technologies; trading strategies with options; and finally the results with related conclusions.

## *2.1 The goal*

As introduced in the volatility section, the IV often does not coincide with the future volatility realized in that period. The market makes errors in these estimates; often they are lawful and rational, but in other cases they are the result of irrational evaluations or impulses; thus going to violate the efficient market hypothesis, resulting in the opportunity for (risk-adjusted) profits above the market return.

There are several strategies that take advantage of this principle, and traditionally they are based on selling or buying volatility, via spreads, when the IV is much above or below the historical average HV. However, such a system is not accurate, and is based on the common idea that The IV should, in the long run, repariate with the average HV. The principle behind the project is to use ML to have a more accurate assessment in the short run.

The neural network, from a set of data (including IV and RV), must classify whether the RV over the next 5 (working) days will be significantly higher or lower than the expected RV (i.e., the estimated 5-day IV). Depending on the assessment, long or short volatility positions will be opened via 4 different types of spreads; in case there is no market inefficiency, no position will be opened.

## *2.2 Dataset used*

The model for its prediction employs various data; before being used, they are, downloaded, selected, processed, scaled and transposed over time.

Specifically, data before being used are processed in 6 stages:

1. Downloads: where they are downloaded;
2. Preprocessing: in which smoothing is done and then making them stationary;
3. Feature selection: where you reduce the number of features while keeping only the most useful ones;
4. PCA: in which the PCA technique is used to remove correlations between features;
5. Scaling: where they are scaled in an interval;
6. Lag and Split: where they are transposed and converted into the final datasets;

### 2.2.1 Downloading

Given the limited availability of data, especially option history; actual histories for Apple stock (AAPL) between 01/01/2011 and 06/01/2022 were used:

The dataset used by the model includes 15 features, plus the target (i.e., classification), they are:

- Price series (*Stock_Price*): Closing price series, adjusted for splits and dividends, downloaded from Yahoo Finance via the appropriate API;
- The past RV series (*RV*): Calculated through a 5-day sliding window, from the price series (for simplicity we exclude dividends);
- The series of past, long-term RV (*RV_Long_Term*): Calculated through a sliding window at 252 days (one working year);
- The IV (*IV*) series: calculated a as a 5-day volatility index (IVX), the strategy adopted for the calculation is explained more fully below;
- Volume series (Volume): Series of the number of shares traded in the market in a day, downloaded from Yahoo Finance via the appropriate API;
- The price-to-earnings (*PE*) rate series: earnings-to-price ratio, obtained by dividing the share price and annualized earnings per

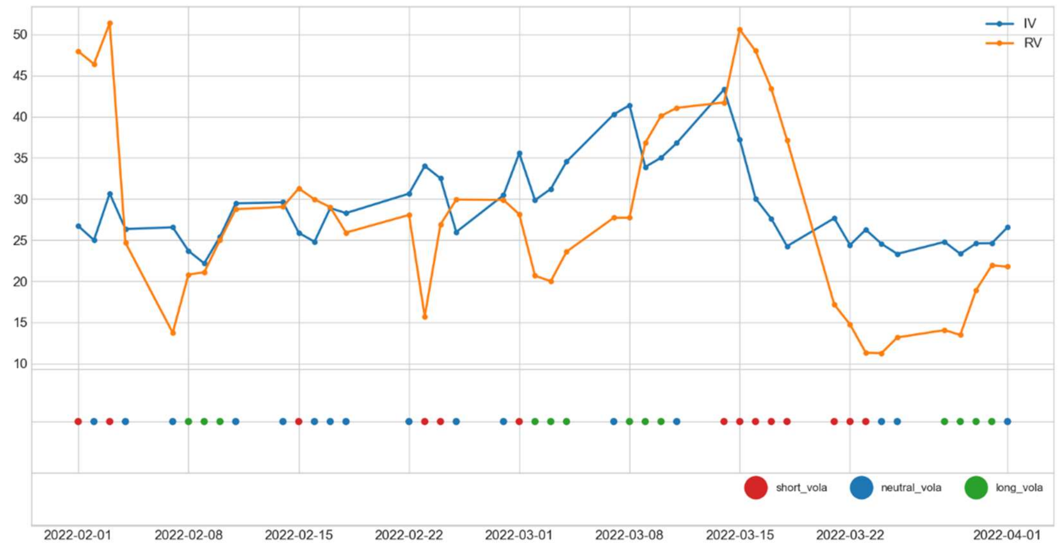
share, the latter of which are downloaded from the Alpha Advantage website via the appropriate API;

- The company interest series (Trend): the interest in terms of Google searches of the related company, downloaded from the GoogleTrends API.
- The series of days missed to reports (*DtE*): i.e., the working days missed to the next quarterly report; the dates are downloaded from the Alpha Advantage website via the appropriate API;
- The series of missing days to dividend (*DtD*): i.e., the business days to the next dividend, the dates are downloaded from Yahoo Finance, via the appropriate API;
- The High Minus Low (HML) series: the first of the three factors of the fame french model, represents the difference between the returns between stocks of value and growth companies;
- The series of Small Minus Large (SMB): the second of the three factors of the fame french model, represents the difference between the returns between stocks of small and large companies;
- The Market Returns Series (MKR): the annualized changes in percentage of the benchmark market index, in this case the one considered is the Nasdaq, also the values are always obtained from the Yahoo Finance API;
- The Market IV series (MK_IV): volatility index of the reference market, due to lack of data the S&P500 and thus the VIX index is considered the market, downloaded from Yahoo Finance via the appropriate API;
- The series of risk free instrument (RFR) returns: 1-month U.S. bond returns, obtained from the Federal Reserve Economic Data (FRED) API;
- The inflation rate series (INFL): 10-year inflation rate, obtained from the Federal Reserve Economic Data (FRED) API;

This dataset can be logically divided into 4 parts: in the first we have information (Stock_price, IV, RV, RV_Long_Term, Volume, Trend) describing the stock's stock market performance; in the second (PE) we have a variable to describe the "health" of the company; in the third (DtE, DtD) we have parameters to estimate the distance from possible catalysts of volatility (it tends to increase in the periods before reports and dividends); finally we have metrics to describe the general state of the market.

The target time series varies between 3 labels: "long," "neutral," and "short"; indicating precisely the expected valuation.

*Figure 1: Series of volatilities and target to rank*



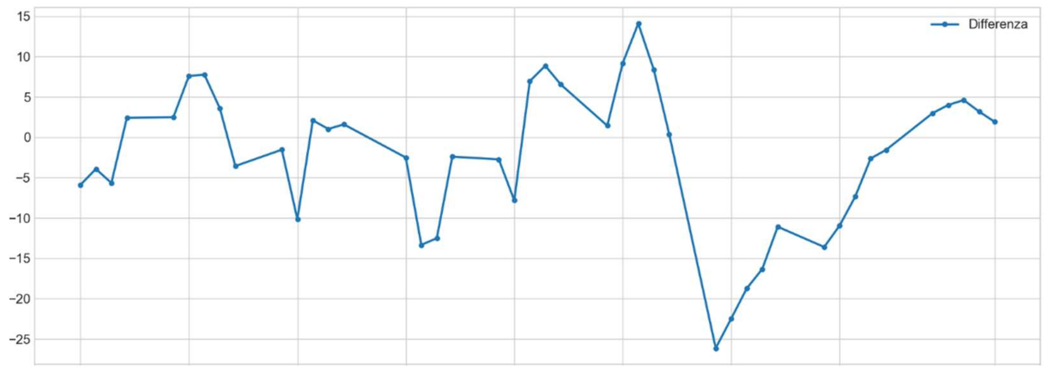


*Figure 2: Difference between implied and realized future volatility. From which one can see the variation of the target with that of the difference*

### *2.2.1.1 Calculation of the volatility index IVX*

The index is volatility, as already mentioned, is used to define the IV of the underlying asset in general. It is necessary because options of the same security with different maturities and strikes have very different IVs, so an indicator that shows the overall set is useful. Currently the calculation of an IVX is done with other derivative instruments, such as variance swaps. Due to the lack of such data I adopt two different strategies, but they are no longer fashionable.

The first strategy is based on the system used by the Chicago Board Options Exchange (CBOE) to calculate the VIX (IVX of the S&P500 at 30g), details of which can be found in the related White Paper [9]; however, it, being native for a 30-day (i.e., 21 business) IVX, is inaccurate on shorter intervals (5g).

The second methodology draws on those adopted by data providers Quantcha, Nasdaq Data Link, and IVolatility , described in [10] e [11]. They differ on some points, but agree on a similar approach. That is: for each day's calculation, the two maturities closest to the date to be estimated are chosen; then, for each, ATM and relatively close options are selected, whose distance-weighted average is calculated; then, the arithmetic average between the value of the calls and the value of the puts is calculated; finally, the interpolation between the two dates is made in order to obtain the approximation for the one in between.

In the project I adopt such a system, choosing linear interpolation between the two dates (the quadratic one although more appropriate, is not feasible due to lack of data), and selecting the ATM strikes and a $\pm 2\%, 4\%, 6\%$; whose weighted average is obtained. It is first calculated linearly, but later, increasing the realism of the indicator I use an exponentially weighted average.

For example, on Wednesday 06/04/2022 you want to estimate the IV at 5g (13/04/2022):

- The two dates closest to the deadline (08/04/2022 and 15/04/2022) are selected.
- On both dates, calls and puts with ATM strikes are selected and at $\pm 2\%, 4\%, 6\%$.
- Exponential averaging is done between the chosen options, separating call and put.
- The arithmetic mean between call and put is calculated, obtaining The estimated IV for 08/04 and 15/04.
- Linear interpolation between the two dates is performed, obtaining the estimated IV for 13/04.

### *2.2.2 Preprocessing*

Non-stationary features are convolved in this step: Stock_Price, Volume, PE, Trend, RFR, INFL). They will first be smoothed (by the smoothing operation) and then become stationary.

The smoothing step is done through the Savitzky-Golay filter, applying light smoothing (set with the 11-day window and 3 polynomial order); so as to reduce noise in the data and make prediction easier.

Stationarity is obtained by calculating the percentage difference of a value with its previous value; for example, the generic feature *x* is obtained as follows: $x_t = \frac{x_t - x_{t-1}}{x_{t-1}}$. This feature, in addition to removing some scaling problems, is essential in some ML models; in the case of LSTM, it is not essential but improves performance.

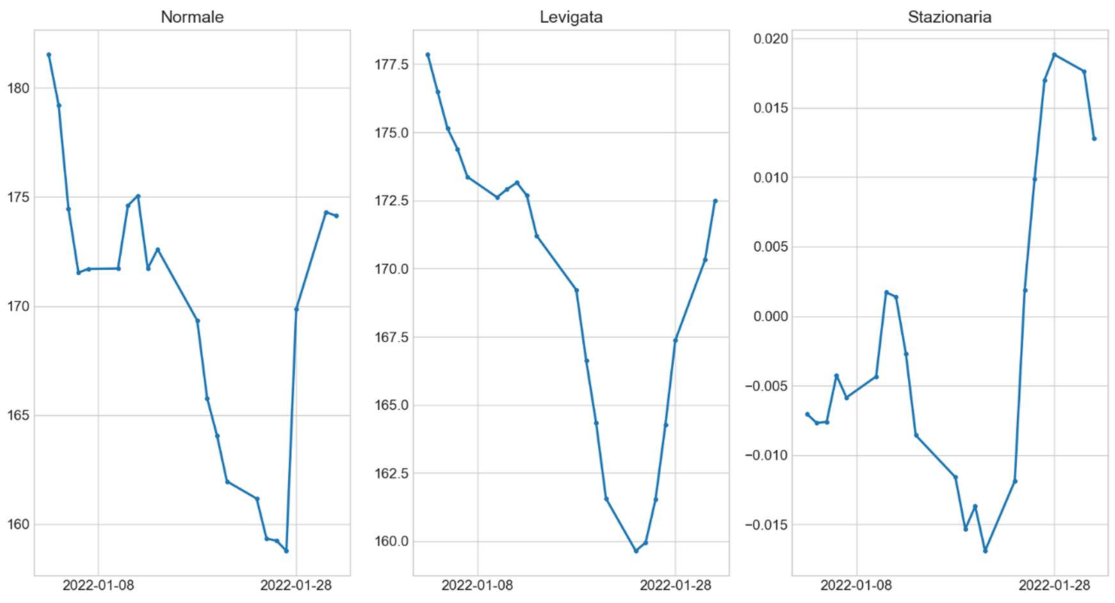


*Figure 3: Price series compared in the two Preprocessing steps.*

### *2.2.3 Feature Selection*

Given the large number of features available, the model could benefit from reduction to the most useful variables; so as to reduce the noise brought by features that correlate poorly with the target, and thus improve training and prediction.

The algorithm defined as *Recursive Feature Elimination* (*RFE*) is used to accomplish this step. It associates each feature with a weight, which determines its importance; this is thanks to an external model, which during training will increase the weights associated with the most relevant features. Then RFE will remove the feature with the lowest weight and proceed, recursively, until the desired amount is reached.

### *2.2.4 PCA*

This phase takes advantage of the principle, which is better explored in [12]; that, the neural network learns more effectively from a few uncorrelated sets than from many correlated ones. In fact, if two variables are highly correlated, they will vary similarly bringing the same information to the network.

First the data will be scaled, via a simple MinMaxScaler; and then, the Principal Component Analysis (PCA) technique is exploited, which eliminates the correlations of the input features; producing uncorrelated series. It is implemented by the function of the same name, from the *scikit-learn* library, details of which can be found in [13].

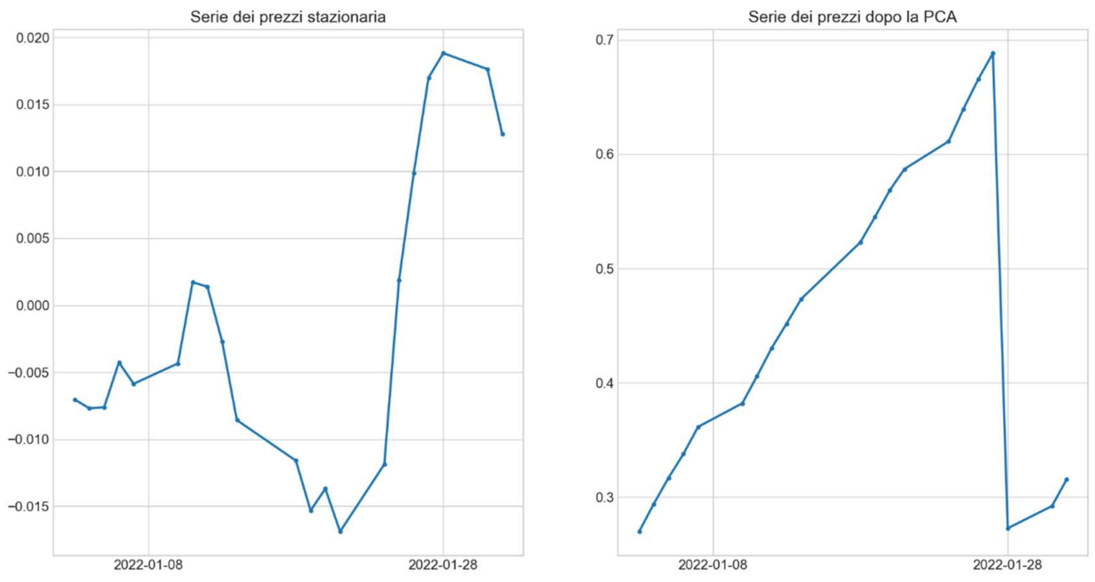


*Figure 4: The effects of PCA on a feature.*

### 2.2.5 Scaling

At this stage the features are scaled into intervals compatible with the network activation functions (-1, +1). While *hot encoding is* applied to the target; that is, the 3 labels are transformed, first into numerical values and then into 3 distinct variables, thus eliminating the closeness relationships between the numbers. For example, the label "neutral" will be transformed into the values [0, 1, 0], eliminating the closeness relationships with the long [1, 0, 0] and short [0, 0, 1] alternatives.

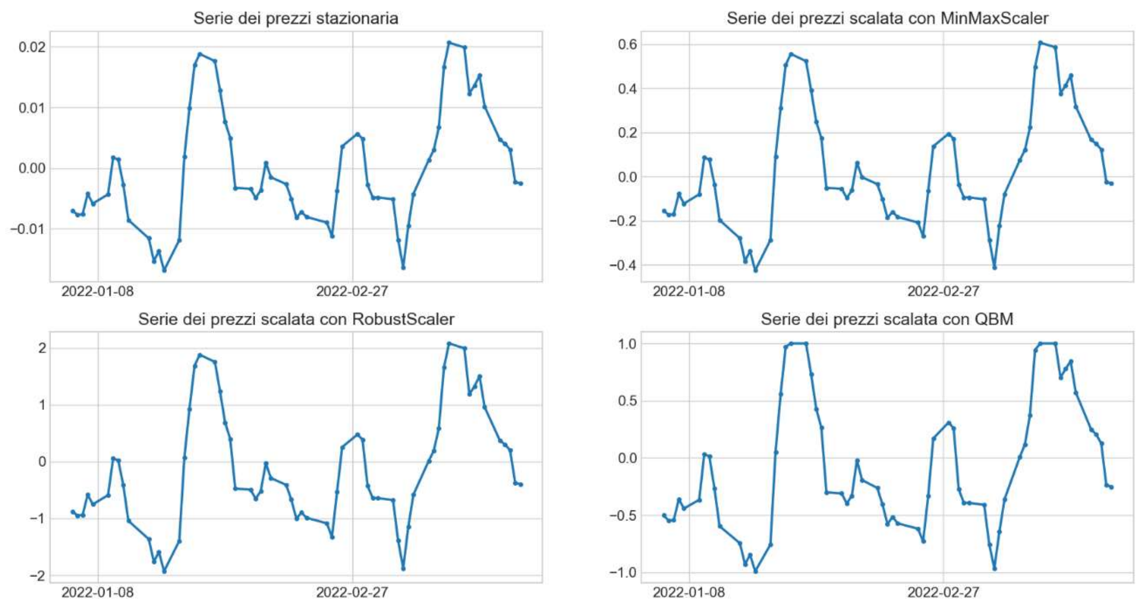
Features are scaled according to the same strategy, chosen from a number of different approaches. This decision is treated as a hyperparameter of the model and therefore depends on the performance obtained on the validation Test. Each approach is implemented via the OOP paradigm: it inherits from an abstract parent class the common interface; and implements its *fit*, *transform*, and *inverse_transform* logics. They are the canonical methods of a scaler object, and handle internal parameter training, data transformation, and inverse transformation, respectively. Each strategy trains its parameters on the train dataset and applies the transformation on the integer dataset. The implemented approaches are:

- Scaler *M* (MinMaxScaler): scales each data in the range (-1, 1):

$$X_{std} = \frac{X - \min(X)}{\max(x) - \min(X)}$$

$X_{scaled} = X_{std} * (\max - min) + min$ *(with max and min extremes of the chosen range)*

- Scaler *R* (RobustScaler): scales data with robust statistics to outliers, i.e., in agreement with quantile intervals
- Scaler *RM*: combines the two scalers, applying first the RobustScaler and then the MinMaxScaler to have the final values in the range (-1, 1)
- *BM* Scaler *(Bound-MinMax)*: applies a transformation to the outliers, sets two thresholds (bounds) at quantiles Q1 and Q3, and reduces the values exceeding them to such, then applies the MinMaxScaler;
- Scaler *QBM* (*QuantileTransformer-Bound-MinMax*): first applies the QuantileTransformer, by which it transforms each data into a normal distribution, respecting outliers, then applies the BM methodology.



## *2.2.6 Lag and Split*

An LSTM network, as discussed in more detail in Chapter 1.2.3.1, uses data arranged in sequences, i.e., in 3 dimensions; therefore, an additional step is needed, in which to transform the time series, into series of sequences of temporally transposed values.

Each feature will then consist of a series of sequences; that is, for each day a feature is formed by the value on that day and previous ones. Instead, the size of the target remains unchanged, since the network will have to produce only one value and not a sequence (it is of the Many-to-one type).

Finally, the dataset is divided, keeping the sorting, into the 3 canonical parts:

- Train Dataset (0-90%): these are the data the program has available; they are used for training the LSTM network and previous tools (scaler, RFE, PCA etc....).
- Validation datasets(90-95%): they are used during the tuning phase of the hyperparameters; that is, we test the model with multiple configurations choosing the one that provides the best performance.
- Test Datasets (95-100%): these are not seen by the model and are used to measure the final accuracy of the model.

## 2.3 Definition of the Model

Several similar models, which have the same components and vary in their arrangement and some hyperparameters, were used for classification. The elements and commonalities will therefore be presented, and then the models used will be described.

### 2.3.1 Activation layers and functions

In each model, 4 types of Layers are adopted according to a common structure: one input; one output; two internal ones repeated several times in pairs with the same order. They are:

- Input Layer: input layer, it does not perform computation and just gets features into the network.
- *Gaussian Noise Layer*: does not perform computation and therefore has no weights to train; adds noise to the data to reduce overfitting, is explained in detail later.
- *LSTM Layer*: layer containing LSTM cells; takes a sequence as input and returns a value or sequence; and other LSTM layers will follow; must return a sequence, otherwise a value.
- *Dense Output Layer*: layer of simple neurons; it performs the final classification, thanks to the use of the *softmax* activation function; it always consists of 3 neurons (because there are 3 possible classes); as with the LSTM layers it is preceded by a Gaussian Noise Layer.

The first two layers do not perform computations, and therefore do not use activation functions; instead, LSTM employs the default functions: tanh for the output result from the cell; sigmoid for recursive activations. While the output layer uses the softmax function, which is necessary for classification.

The standard approach to modeling a classification problem is to predict the probability that an element belongs to a class; this solves the classification as if it were predicting an actual value. There are several candidate functions for this task (such as max, argmax and sigmoid), however, they are only usable in one-class binary problems. In this case, having three classes, sigmoid is used because it returns an array of membership probabilities, the total sum of which is always 1 (i.e., 100% of the probability set).

### 2.3.2 Loss function and metrics

The loss function used is *Crossentropy,* the use of which is described in detail in [14]. Basically, it is a measure of the difference between two probability distributions. In particular, the *Categorical* variant is used, which is specific to multi-label classification problems. So, the model is trained with the goal of minimizing that function.

Cross-Entropy Loss:

$$L = \sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

Where:

- *M*: is the number of classes;
- o: is the observation in question
- $y$: is a binary indicator (0,1) that indicates whether *c* is the correct classification for o
- $p$: is the observed probability that o is of class *c*
- The formula corresponds to a single instance; the total is given by the average

The metric used is *Accuracy*, in the declination CategoricalAccuracy. It is a very simple function that calculates the frequency in which the classes predicted by the network coincide with the actual classes.

### 2.3.3 Generalization strategies

To make the models more generalist, I adopted 4 different strategies: dropout; introducing noise with Gaussian Noise Layers; Early Stopping; and Pretraining. They act in the Training phase of the network, with the goal of making the model more generalist without reducing its complexity.

Dropout is a regularization strategy that is part of the Ensemble Methods family. In this strategy, applied to a layer with a parameter $\rho$; multiple networks are trained in parallel, in which some output connections will be canceled (dropped out) with a probability $\rho$, finally maintaining the optimal one. By introducing such a random effect, the network cannot rely too much on each node, and is therefore forced to be more general.

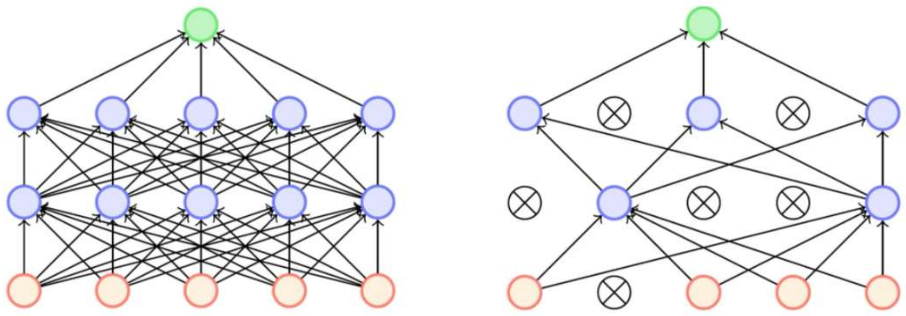


Figure 5: Dropout example. On the left the normal network, on the right one after dropout

Gaussian Noise Layers are layers that add "noise" to the input data and then pass it to the next layer. It is taken from a Gaussian distribution with variance defined as the hyperparameter of the layer. This reduces overfitting because the network cannot "store" the passed random component.

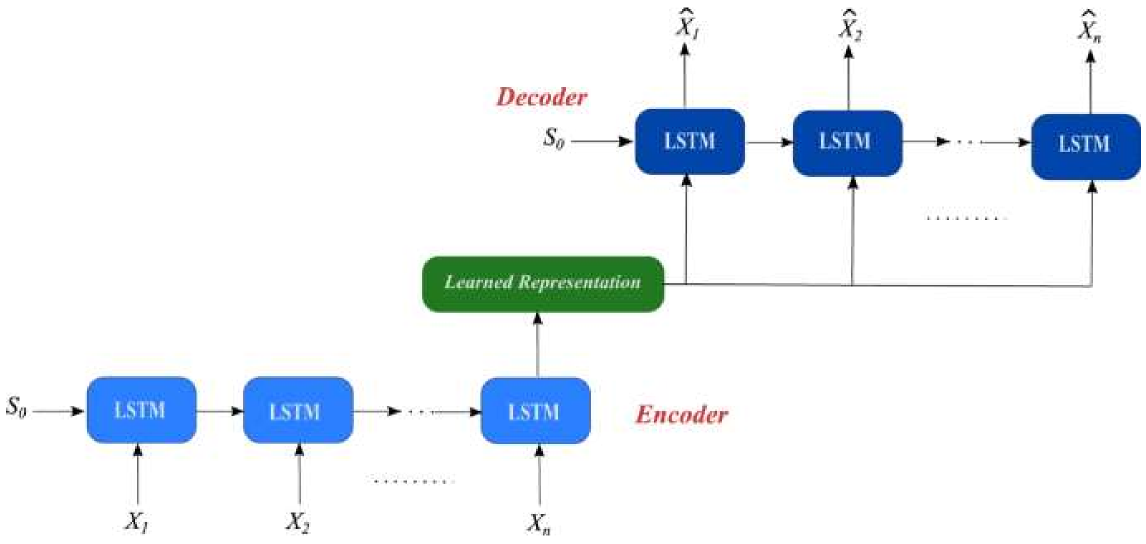
Early Stopping is a technique that relies on stopping the training process early, when the loss function (or a chosen metric) computed on the validation dataset, does not undergo improvement after a predetermined number of epochs. In this case. a tool from the keras. library is exploited to restore the network from the last point of improvement.

#### 2.3.3.1 Pretraining in detail

Pretraining is a step, in this case unsupervised, that anticipates training and is used to initialize the weights of the neurons; so as to avoid areas of local minima, and thus making the model more generic. The general idea is to train an unsupervised model with only the input features, and then transfer the weights into the final model.

The implementation of this practice exploits an LSTM *Autoencoder*, i.e., a network composed of multiple layers of LSTMs; whose cells first decrease in quantity, until they reach a single core, and then reverse the process until the original layer returns. For example, a 2-layer autoencoder with 4 features will have: 4 nodes in the first layer; 2 nodes in the second layer; 1 node as core;

2 nodes in the third layer; 4 nodes in the fourth layer. During its training, the core and middle layers "store", with less and less weights, a reduced representation of the same data; and this makes future training of the network more efficient.



The implemented procedure follows the following logical flow:

1. Only LSTM layers that return a sequence, to be initialized, are copied from the initial model.
2. The autoencoder is constructed; that is:
    2.1. The core (consisting of a single LSTM cell) is added.
    2.2. You copy the initial structure with an inverted (mirror) symmetry.
    2.3. Finally, an LSTM layer with number of cells equal to the features is added, so that the input and output dimensions coincide.
3. You train the autoencoder in an unsupervised manner with data from the train dataset.
4. Computed weights are taken from the original layers, to initialize the model layers.

### *2.3.4 Models adopted*

Three models were used; they differ in the number of layers, nodes, degree of dropout, and use of Gaussian Noise. These models are compared with each other, and with themselves by varying the hyperparameters. Each model is then trained with sequences of lengths 20 and 60; for 300 and 500 epochs; with the use of each implemented scaler (M, R, RM, BM and QBM); with and without the use of feature selection.

Each model follows common parameters:

- in case of feature selection the number of units in the first layer is halved;
- batch size is 32;
- the smoothing window is 11;
- each LSTM layer, which is followed by another LSTM, returns a sequence;
- PCA is always employed;
- Early Stopping epochs are equal to one third of the training epochs;
- Pretraining epochs are equal to one quarter of the training epochs;
- I input and output layers are identical for each model;
- Each layer (except the input layer) is preceded by a Gaussian Noise layer.

Specifically, the inner layers are:

- The first model has the following internal layers:

| ***Layer*** | **Unit** | **Dropout** | **Recurrent dropout** | **Gaussian Noise** |
|---|---|---|---|---|
| *1st layer LSTM* | 30 | 0.2 | 0.2 | 0.2 |
| *2nd layer LSTM* | 8 | 0 | 0 | 0 |

- The second model has the following internal layers:

| ***Layer*** | **Unit** | **Dropout** | **Recurrent dropout** | **Gaussian Noise** |
|---|---|---|---|---|
| *1st layer LSTM* | 15 | 0.2 | 0.1 | 0.2 |
| *2nd layer LSTM* | 8 | 0.1 | 0.1 | 0.1 |

| | | | | |
|---|---|---|---|---|
| *3rd layer LSTM* | 5 | 0 | 0 | 0 |

- The third model has the following internal layers:

| ***Layer*** | **Unit** | **Dropout** | **Recurrent dropout** | **Gaussian Noise** |
|---|---|---|---|---|
| *1st layer LSTM* | 30 | 0.2 | 0.1 | 0.2 |
| *2nd layer LSTM* | 8 | 0.2 | 0.2 | 0.2 |
| *3rd layer LSTM* | 5 | 0 | 0 | 0 |

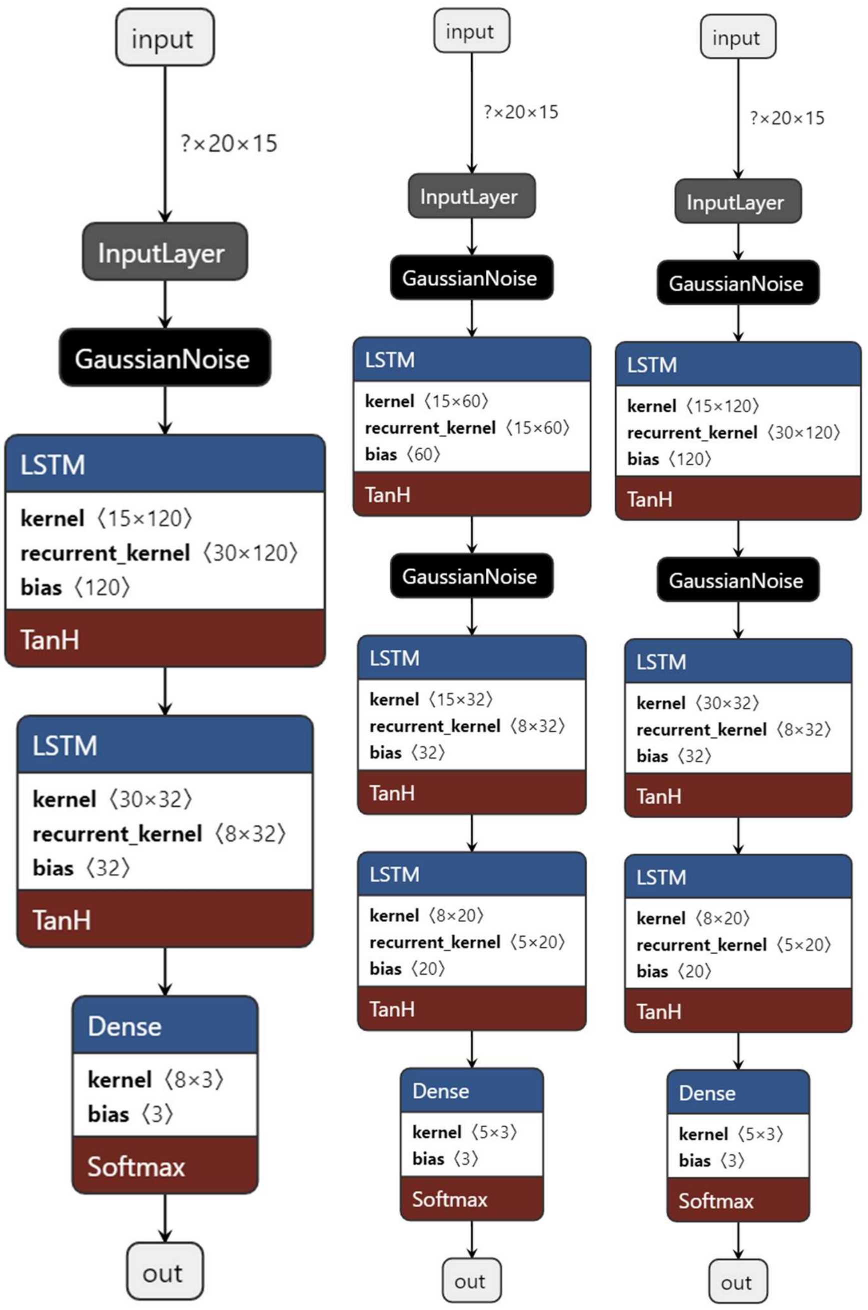


*Figure 6: The first, second and third models, respectively.*

## 2.4 Trading Strategy

An explanation of the trading system will follow in this section, delving into the methodology employed and the spreads adopted, delving into their maintenance over the life time.

### 2.4.1 General operation

As previously mentioned, the goal of the network is to provide a buy, sell or null order to the trading system; depending on the outcome the resulting position will be taken, with the help of 4 spreads that will be compared.

Spreads will be opened on Friday at the close (given the availability of only the closing data), expiring one week and held until that. They are of 4 types, and are all designed for exposure to implied volatility. Therefore they have:

- neutral delta;
- Positive vega if the spread is long at volatility, negative otherwise;
- negative theta if the spread is short to volatility, otherwise positive.

An order is null, if the difference between IV and RV does not exceed a threshold, equal to a percentile chosen and calculated over the previous 125 days.

### 2.4.2 The spreads adopted

We can divide spreads into 2 binary categories; depending on the options, also called *Leg,* that make them up: simple ones, consisting of 2 options; and advanced ones, consisting of 4 options. They are:

- *Straddle*: composed of a call and a put, at the same maturity and same ATM strike, has a generally high entry debt or credit; if it comes:
  - Bought: through the purchase of the two Legs, you get a long volatility exposure with an entry debt, has limited risk and unlimited profit;

    - Sold: through the purchase of the two Legs, you get short exposure to volatility with an entry credit, has unlimited risk and limited profit;

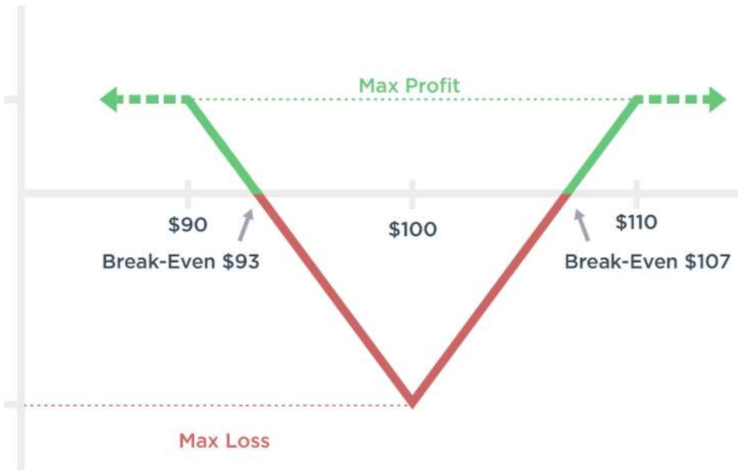


*Figure 7: Long straddle*

- *Strangle*: composed of a call and a put, at the same maturity and at two symmetrical OTM strikes, has a lower entry debt or credit than the straddle; if it comes:
    - Bought: through the purchase of the two Legs, you get a long volatility exposure with an entry debt, has limited risk and unlimited profit;
    - Sold: through selling the two Legs, you get short exposure to volatility with an entry credit, has unlimited risk and limited profit;

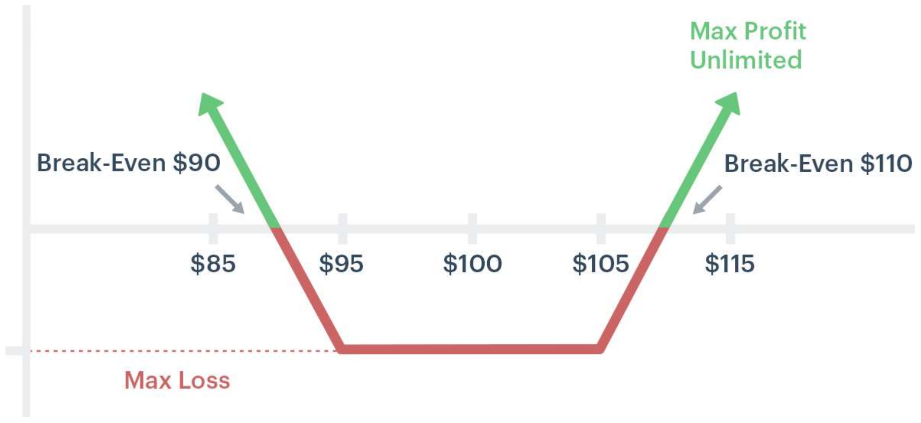


*Figure 8: Long strangle*

- *Butterfly Call*: composed of four calls, at the same maturity; two sold ATM, two bought at two symmetrical strikes one ITM and the other OTM; if it comes:
    - Buy: you get short exposure to volatility, with entry debt, has limited risk and profit;
    - Sold: by reversing the previously described positions, a long volatility exposure is obtained, with an entry credit, has limited risk and profit;

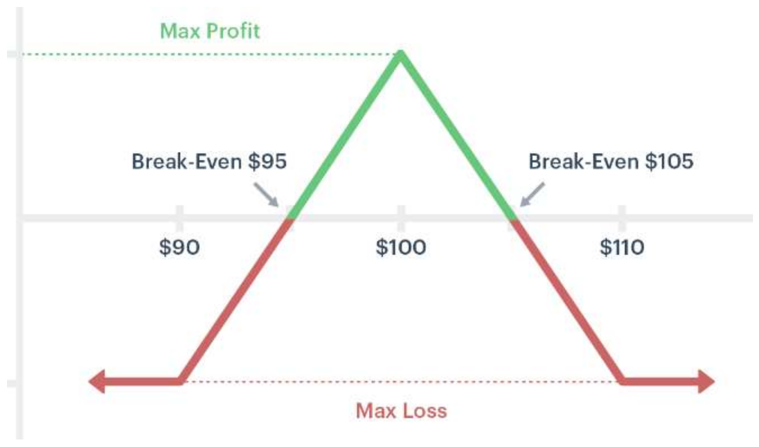


*Figure 9: Long Butterfly*

- *Iron Condor*: composed of two pairs of a call and a put with symmetrical strikes and at the same maturity; the first has OTM strikes and is sold, the second has more distant OTM strikes and is bought; if you:
    - Buy: you get short exposure to volatility, with an entry credit, has limited risk and profit;
    - Sells: by reversing the previously described positions, you get long exposure to volatility, with entry debt, has limited risk and profit;

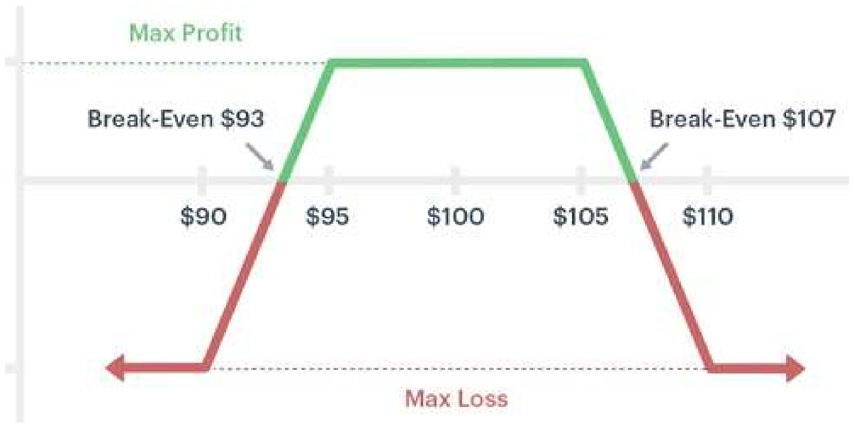


*Figure 10: Long Iron Condor*

So the first two spreads have lower commissions costs (because there are fewer options to trade), however, they do not have potentially unlimited losses. In contrast, spreads such as long strangle and short iron condor have lower entry costs but require larger movements to go into profit; conversely short strangle and long iron condor, require smaller movements but bring less profit.

The implementation is done through the use of a *SpreadLeg* class, which depending on the position and option (call or put) fetches the correct data from the dataset; and a *Spread* class that defines the interface and common methods of spreads (such as the plot and hedging function), Finally, each spread inherits the generic class, and implements the abstract methods by which strikes and binds are calculated.

### 2.4.3 Delta hedging

These spreads, although bought with neutral delta, take on a positive or negative delta the more the price continues to move from the initial value at which it was bought. This property is described by the range of the spread, which in these cases is not neutral. This implies that the value of the spread will be affected by the direction of movement.

To solve this problem, one can close the position (profit or loss) prematurely, but this would go against the goal of the project. The solution adopted is to hedge the position.

Hedging involves buying or selling the underlying asset, equal to the assumed delta of the position at that time; when it exceeds a certain threshold (in this case 30). Specifically, one buys when the delta is negative and sells when it is positive, in order to even out the exposure to the delta. This again makes the spread neutral to the direction of movement, thus leading to reduced risk, resulting in lower profits.

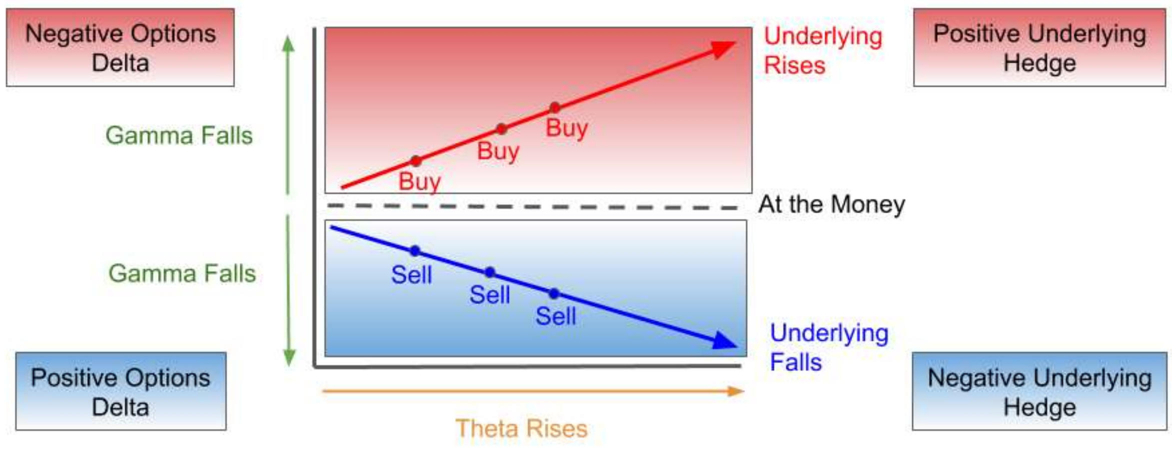


*Figure 11: Short straddle with delta hedging.*

## 2.4.4 Some examples

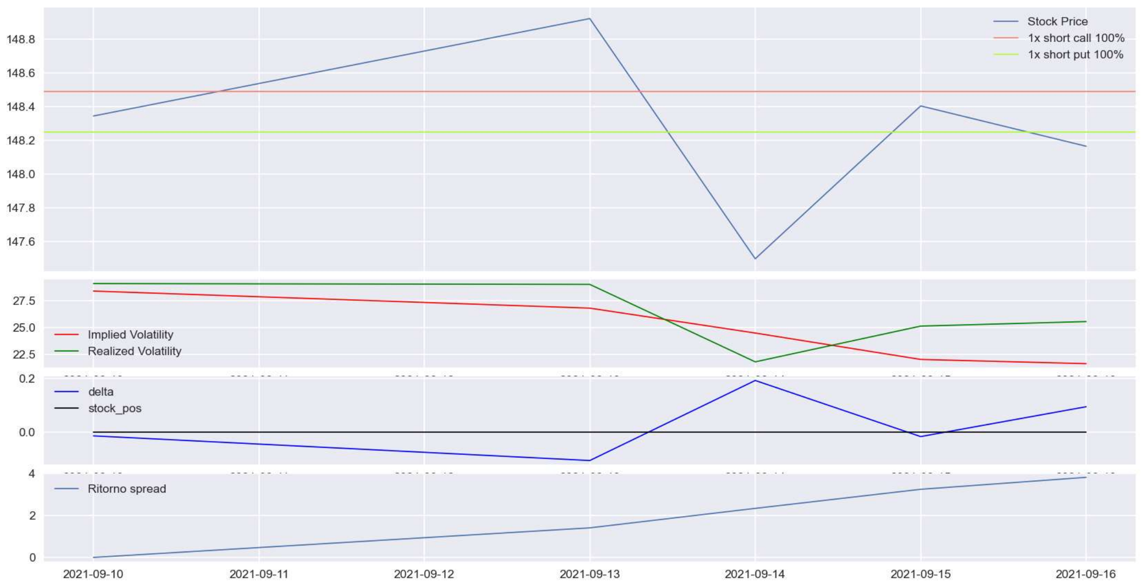


*Figure 12: Example of a calculated short straddle. In this case, the expected IV is not realized, and therefore the spread goes into profit. Also, hedging is not used because the delta does not exceed the minimum threshold.*

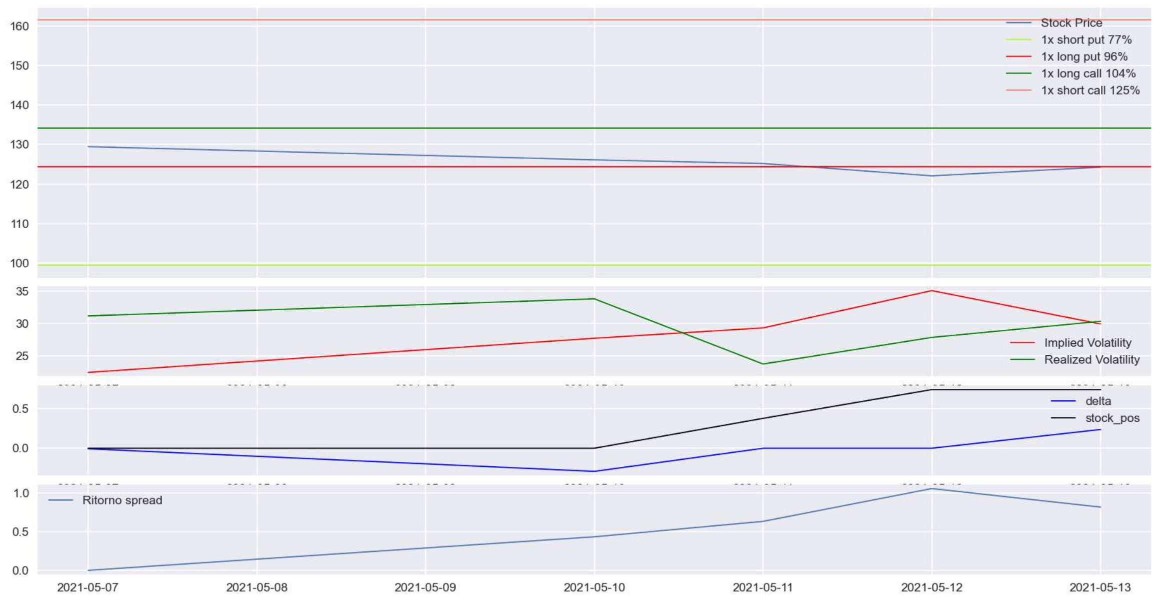


*Figure 13: Example of calculated Short Iron Condor. In this case, implied volatility is exceeded by realized volatility, resulting in a profit. However, delta hedging is unnecessary and reduces the profit made from the spread.*

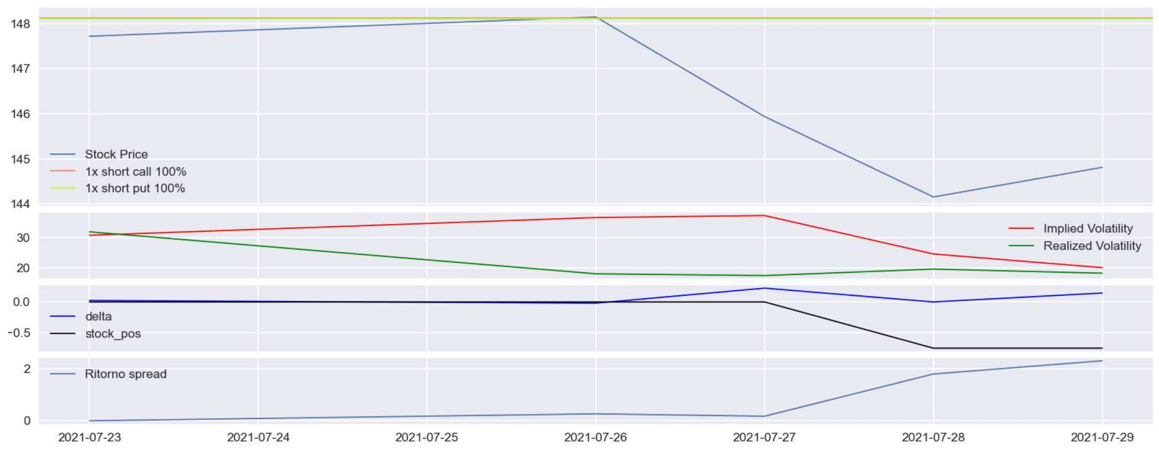


*Figure 14: Example of a calculated short straddle. In this case the implied volatility is not realized, but given the price movement developed, this spread would still be at a loss. Thanks to hedging, the delta remains neutral and the spread ends in profit.*

### 2.4.5 Selection of spread strikes

Strikes, as defined earlier, are composed of one or two pairs of options that the same distance from the current price; which must be established a priori and is influenced by the hedging frequency and percentile selected. To make this choice, candidate values are selected first; then the last 20 percent of the train dataset is used to elect the best values.

## 2.5 The results

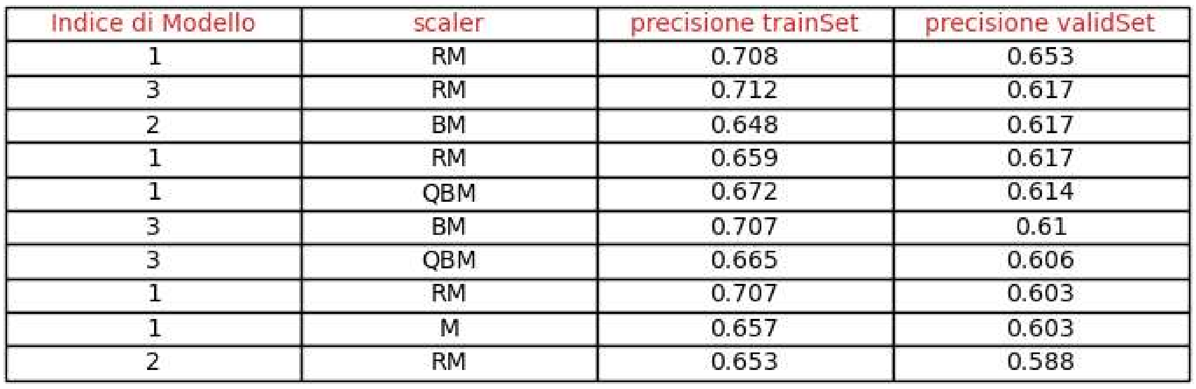
This section will show the results obtained from the selection processes and tests. Thus visualizing the optimal models and spreads, then ending with the final result.

### 2.5.1 Results of the models

The best results were generally obtained with:

- 500 training eras;
- The use of feature selection;
- sequences with a length of 20

The rest of the hyperparameters were tested thoroughly, with the following results:



| Indice di Modello | scaler | precisione trainSet | precisione validSet |
|---|---|---|---|
| 1 | RM | 0.708 | 0.653 |
| 3 | RM | 0.712 | 0.617 |
| 2 | BM | 0.648 | 0.617 |
| 1 | RM | 0.659 | 0.617 |
| 1 | QBM | 0.672 | 0.614 |
| 3 | BM | 0.707 | 0.61 |
| 3 | QBM | 0.665 | 0.606 |
| 1 | RM | 0.707 | 0.603 |
| 1 | M | 0.657 | 0.603 |
| 2 | RM | 0.653 | 0.588 |

*Figure 15: The results of the hyperparameter tests.*

### 2.5.2 Results of spreads

Each result is the compounded percentage return of the strategy. That is, first the simple return of each position is calculated and then their composition is calculated. However, these values are useful only for comparative purposes; because they do not consider commissions and capital required on margin for short trades.

Simple return of a spread:

$$r = \frac{P - V}{|P|} + r_s$$

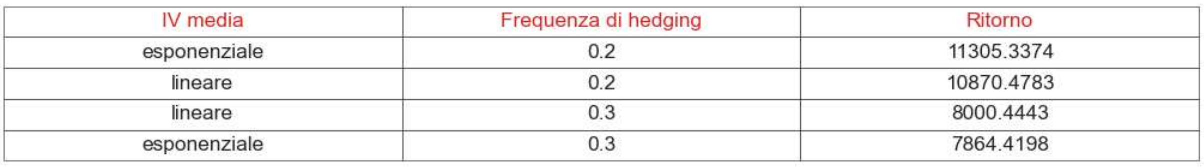
Where:

- *P*: is premium of the spread (if it is a credit it is positive, otherwise it is negative)
- *V:* is the final price of the spread
- $r_s$: is the compound return of the stock for the retained portion (caused by hedging)

Total return is the composition of simple returns throughout the trading period.

The calculation method for IV, hedging frequency and strike offsets will be compared; in order to determine the best parameters.

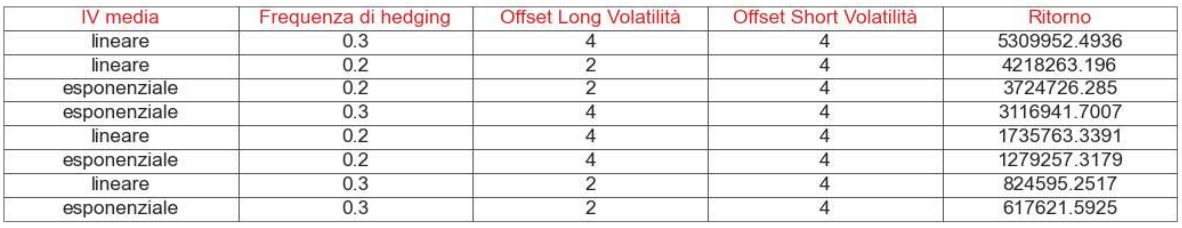
### *2.5.2.1 Straddle:*



| IV media | Frequenza di hedging | Ritorno |
|---|---|---|
| esponenziale | 0.2 | 11305.3374 |
| lineare | 0.2 | 10870.4783 |
| lineare | 0.3 | 8000.4443 |
| esponenziale | 0.3 | 7864.4198 |

*Figure 16: The best returns for the Straddle.*

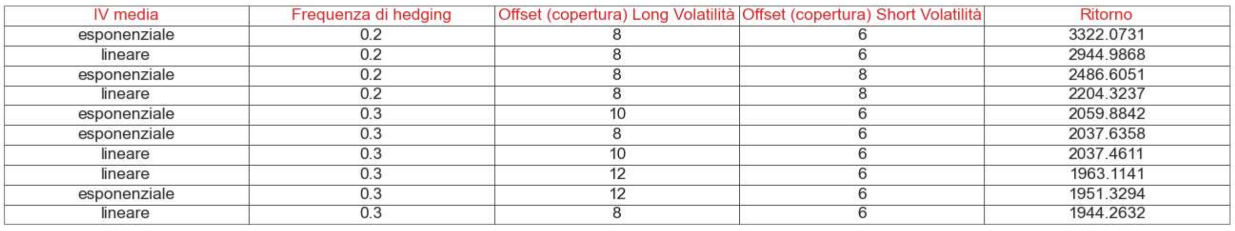
### *2.5.2.2 Strangle:*



| IV media | Frequenza di hedging | Offset Long Volatilità | Offset Short Volatilità | Ritorno |
|---|---|---|---|---|
| lineare | 0.3 | 4 | 4 | 5309952.4936 |
| lineare | 0.2 | 2 | 4 | 4218263.196 |
| esponenziale | 0.2 | 2 | 4 | 3724726.285 |
| esponenziale | 0.3 | 4 | 4 | 3116941.7007 |
| lineare | 0.2 | 4 | 4 | 1735763.3391 |
| esponenziale | 0.2 | 4 | 4 | 1279257.3179 |
| lineare | 0.3 | 2 | 4 | 824595.2517 |
| esponenziale | 0.3 | 2 | 4 | 617621.5925 |

*Figure 17: The best returns for the Strangle.*

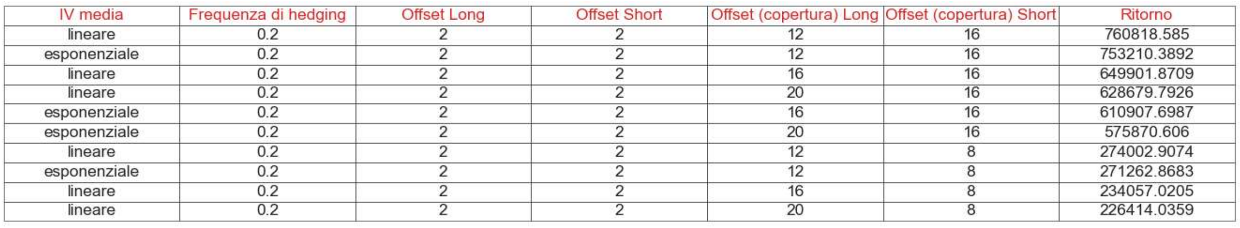
### 2.5.2.3 Butterfly:



| IV media | Frequenza di hedging | Offset (copertura) Long Volatilità | Offset (copertura) Short Volatilità | Ritorno |
|---|---|---|---|---|
| esponenziale | 0.2 | 8 | 6 | 3322.0731 |
| lineare | 0.2 | 8 | 6 | 2944.9868 |
| esponenziale | 0.2 | 8 | 8 | 2486.6051 |
| lineare | 0.2 | 8 | 8 | 2204.3237 |
| esponenziale | 0.3 | 10 | 6 | 2059.8842 |
| esponenziale | 0.3 | 8 | 6 | 2037.6358 |
| lineare | 0.3 | 10 | 6 | 2037.4611 |
| lineare | 0.3 | 12 | 6 | 1963.1141 |
| esponenziale | 0.3 | 12 | 6 | 1951.3294 |
| lineare | 0.3 | 8 | 6 | 1944.2632 |

*Figure 18: The best returns for the Butterfly.*

### 2.5.2.1 Iron Condor:



| IV media | Frequenza di hedging | Offset Long | Offset Short | Offset (copertura) Long | Offset (copertura) Short | Ritorno |
|---|---|---|---|---|---|---|
| lineare | 0.2 | 2 | 2 | 12 | 16 | 760818.585 |
| esponenziale | 0.2 | 2 | 2 | 12 | 16 | 753210.3892 |
| lineare | 0.2 | 2 | 2 | 16 | 16 | 649901.8709 |
| lineare | 0.2 | 2 | 2 | 20 | 16 | 628679.7926 |
| esponenziale | 0.2 | 2 | 2 | 16 | 16 | 610907.6987 |
| esponenziale | 0.2 | 2 | 2 | 20 | 16 | 575870.606 |
| lineare | 0.2 | 2 | 2 | 12 | 8 | 274002.9074 |
| esponenziale | 0.2 | 2 | 2 | 12 | 8 | 271262.8683 |
| lineare | 0.2 | 2 | 2 | 16 | 8 | 234057.0205 |
| lineare | 0.2 | 2 | 2 | 20 | 8 | 226414.0359 |

*Figure 19: The best returns for the Iron Condor.*

### **2.5.3 Final result**

The final system will use the best-performing model and optimal spread settings. Obtaining:

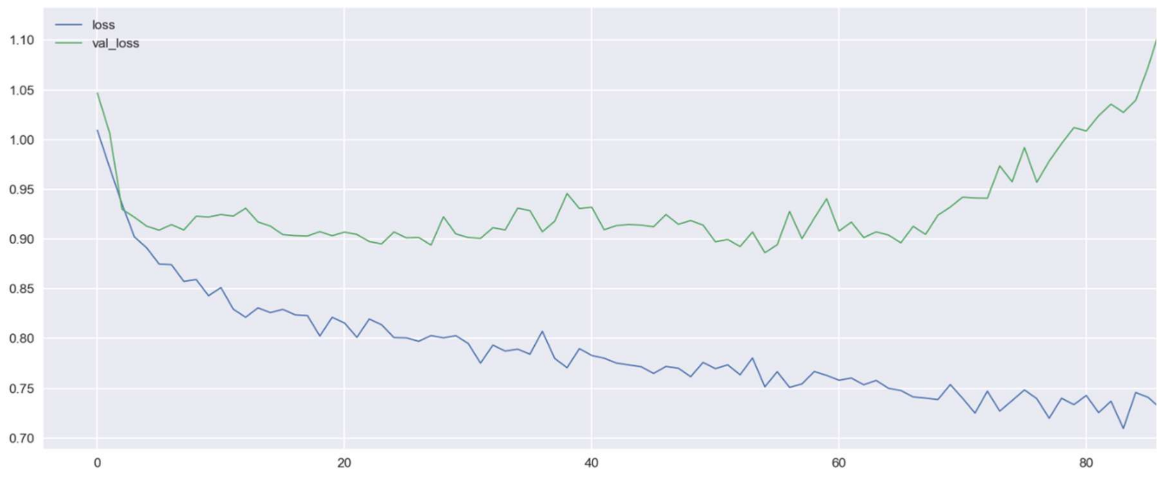


*Figure 20: Training history. As can be seen from the loss of validation (in green), after a very early phase of improvement the network stops learning patterns.*

*Figure 21: TrainSet prediction. (Accuracy: 71%)*

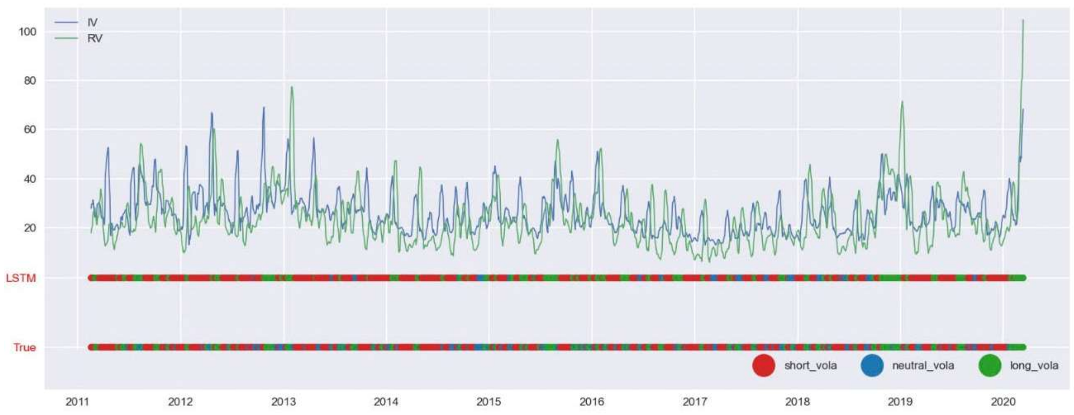
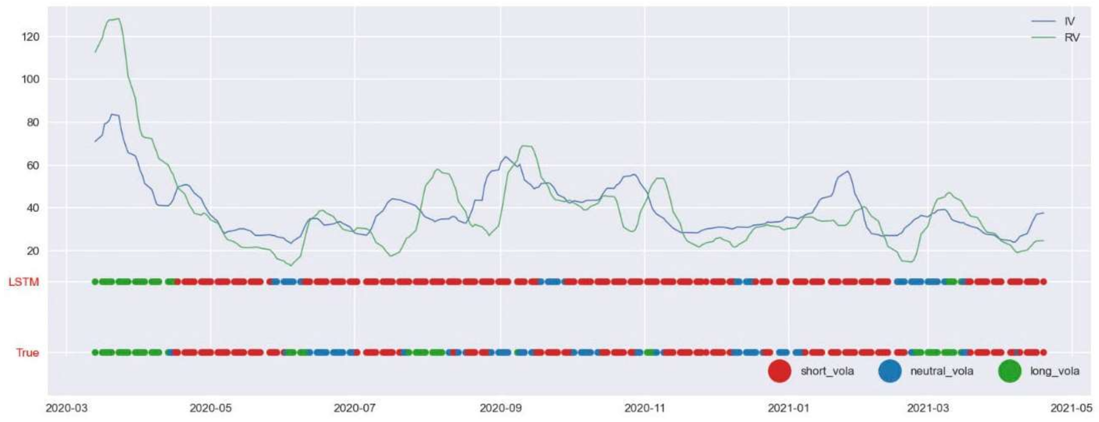


*Figure 22: Prediction on ValidationSet. (Accuracy: 65%)*



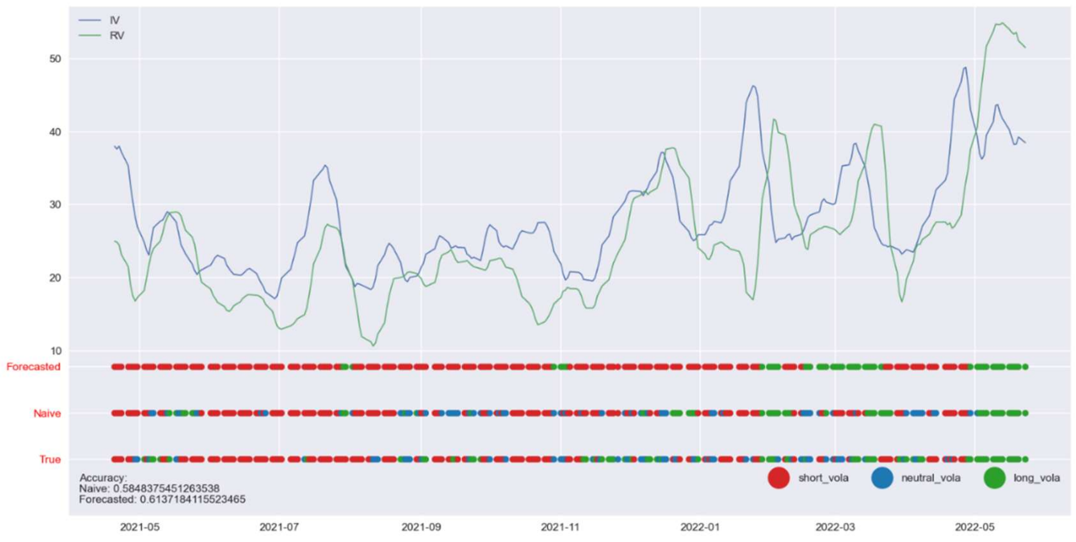


*Figure 23: Forecast on TestSet.(Accuracy of 61%) Forecast results are compared with actual values and Naive Forecast*

The performance of the system is compared with an unintelligent algorithm, the so-called *Naive Forecast*. It is realized without the ML; therefore, our model becomes useless if it cannot beat it. In this case, the Naive Forecast is made with the basic strategy; the one in which we buy or sell volatility if the difference between IV and RV exceeds a certain historical threshold (usually the mean).

As can be seen, the model fails to pass the Naive Forecast. This means that the Neural Network fails to learn patterns in the data, and while there may be many causes, one of the most likely is related to the *Random Walk* theory.

Finally, applying the obtained forecasts, the system makes profits for the first few weeks; but as soon as the wrong forecasts are reached, the system makes losses bringing the total return to negative.

#### *2.5.3.1 Random Walk Hypothesis*

This is a popular hypothesis and the center of research and debate; although it is typically considered true, at least from a theoretical point of view.

This theory, deepened and tested in [15], states that it is not possible to predict future price changes, from past ones; because, each piece of information is assimilated into the market randomly, resulting in random deviations. Otherwise, if deviations were systematic, it would be possible to make arbitrages; but, these would eventually cancel out in the long run. So, the price series can be associated with a random movement and therefore not predictable with ML.

The subjects of the project were implied and realized volatility; they by definition, are the standard deviation of price changes (one expected, the other obtained). This connection could mean that Random Walk theory indirectly extends to volatility as well, at least in the short term. A very interesting future development could deal with investigating and testing the existence of this link.

# Conclusions

Machine Learning, and neural networks in particular, are a very powerful tool. In this paper, a summary view of their characteristics has been taken; from which one can see both the weaknesses and strengths, but most importantly the potential of these tools.

Finance is only one of the possible application domains for ANNs, and as highlighted in [16], the number of papers on the subject, and thus the related interest, is continuously growing; however, most of them are oriented toward forecasting prices (of stocks and indices) and trends. This paper delves into a topic that is not yet too much involved: that of volatility, and proposes a practical solution. It also places special interest on the use of real and not simulated data; which usually simplify forecasts, making them unrealistic.

As pointed out in the results section, despite the various means employed, the proposed solution fails to consistently overcome the Naive forecast. Future developments may investigate the failure of this attempt; demonstrating the possible link of volatility with Random Walk theory, and how this makes the approach used inapplicable. Other developments may instead propose new solutions: isolating one of the two volatilities, limiting it to a single forecast; or applying other technologies, such as reinforcement learning.

## Bibliography:

[1] G. Rebala, A. Ravi, and S. Churiwala, An Introduction to Machine Learning, Springer Publishing Company, 2019.

[2] M. Mohri, A. Rostamizadeh, and A. Talwalkar, Foundations of Machine Learning 2, Cambridge (MA): MIT Press, 2018.

[3] C. Aggarwal, Neural Networks and Deep Learning: A Textbook (1st. ed.), Springer Publishing Company, Incorporated, 2018.

[4] B. Krose and P. Van der Smagt, An introduction to neural networks, 2011.

[5] A. F. Murray, Applications of neural networks, Boston: Kluwer Academic Publishers, 1995.

[6] K. Mehrotra, C. K. Mohan and S. Ranka, Elements of artificial neural networks, MIT press, 1997.

[7] Y. Yong, S. Xiaosheng, and Z. Jianxun, "A review of recurrent neural networks LSTM cells and network architectures," in *Neural Computation*, 2019, pp. 1235-1270.

[8] C. Olah, "Understanding LSTM Networks," 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs.

[9] CBOE, "VIX White Paper," [Online]. Available: https://cdn.cboe.com/resources/vix/vixwhite.pdf.

[10] N. D. L. Quantcha, "US Equity Historical & Option Implied Volatilities," [Online]. Available: https://data.nasdaq.com/data/VOL-us-equity-historical-option-implied-volatilities/documentation?anchor=knowledge-base.

[11] IVolatility, "IVolatility Education - Implied Volatility Index (IV Index)," [Online]. Available: https://www.ivolatility.com/help/5.html.

[12] B. Boukhatem, S. Kenai, A. T. Hamou, D. Ziou, and M. Ghrici, "Predicting concrete properties using neural networks(NN) with principal

component analysis(PCA) technique," Techno-Press, P. O. Box 33 Yusong Taejon 305-600 Korea, 2012.

[13] scikit-learn, "Principal component analysis (PCA)," [Online]. Available: https://scikit-learn.org/stable/modules/decomposition.html#pca.

[14] G. E. Nasr, E. Badr, and C. Joun, "Cross entropy error function in neural networks: Forecasting gasoline demand," in *FLAIRS conference*, 2002, pp. 381-384.

[15] V. Horne, J. C. Parker and G. C. Parker, "The Random-Walk Theory: An Empirical Test," in *Financial Analysts Journal, vol. 23*, 1967, pp. 87-92.

[16] O. B. Sezer, M. U. Gudelek, and A. M. Ozbayoglu, Financial time series forecasting with deep learning: A systematic literature review: 2005-2019, 2020.

[17] A. Krogh, "What are artificial neural networks?," Nature Biotechnology, 2008, pp. 195-197.

[18] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," in *Chemometrics and Intelligent Laboratory Systems*, Elsevier Science, 1997, pp. 43-62.

[19] H. M. Sazli, A brief review of feed-forward neural networks, Communications Faculty of Sciences University of Ankara Series A2-A3, Physical Sciences and Engineering 50, 2006.

[20] D. Erhan, A. Courville, Y. Bengio, and P. Vincent, "Why Does Unsupervised Pre-training Help Deep Learning? *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics,*" in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Teh, Yee Whye and Titterington, Mike; PMLR, 2010, pp. 201-208.

[21] F. Wang and D. M. J. Tax, Survey on the attention based RNN model and its applications in computer vision, ArXiv abs/1601.06823, 2016.

[22] W. C. Merrill, W. Gail, Y. Goldberg, R. Schwartz, A. N. Smith, and E. Yahav, A Formal Hierarchy of RNN Architectures, ACL, 2020.

[23] M. Schuster and K. Paliwal, "Bidirectional recurrent neural network," in *Transactions on Signal Processing*, IEEE, 1997, pp. 2673-2681.

[24] C. Bennett, Trading Volatility: Trading Volatility, Correlation, Term Structure and Skew, 2014.

[25] L. Downey, "Essential Options Trading Guide.," 2022. [Online]. Available: https://www.investopedia.com/.