Institut de Statistique, Biostatistique et Actuariat

Neural networks for pricing derivatives on temperatures

Membres du jury:

Prof. D. Hainaut *Promoteur*
Prof. P. Devolder

Mémoire présenté en vue de
l'obtention du mastère
en sciences actuarielles
(orientation sciences actuarielles)
par:

Rosseels Guillaume

August 16, 2021

Academic year 2020-2021

Preface

This dissertation has been prepared in partial fulfillment of the requirements for the master degree in actuarial sciences delivered by the Université Catholique de Louvain.

Acknowledgment

Firstly, I would like to express my gratitude to Donatien Hainaut for advising and supporting me during this thesis. His experience allowed me to develop my ideas but also to be able to put them into practice.

I also thank Mr. Pierre Devolder, the reader of my thesis, for the time spent reading this work.

I also thank my friends for their precious advice and support during the writing of this thesis but more generally during my years of study.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Weather derivatives have a payoff which depends directly on the value of an underlying weather index, whose purpose is to represent weather conditions over time. They are therefore based on temperatures, precipitations or other weather variables. Like many other financial derivatives, these can be used for speculative purposes as well as for hedging. Therefore, these products allow different agents to protect themselves or at least to reduce the volatility of their results when these depend on climatic conditions. Moreover, climate risk is often cited as one of the major risks for the years to come by various professionals. However, there also exists insurance that covers climatic risk but these products present some differences. Firstly, the purpose of an insurance is to protect (from a financial point of view) the buyer against a rare and catastrophic event where the weather derivative will protect against recurrent/cumulative and unfavorable conditions. Secondly, in the insurance case, the payment is conditional on the existence of the damage and its amount whereas for weather derivative the payoff structure is based on index and does not depend on the existence of a damage or its amount.

One of the main drivers for the development of this type of product is the energy industry. Indeed, the demand for energy depends strongly on weather conditions, for example, a winter that is not as cold as usual would probably result in a decrease in the demand for energy and therefore in the price of it. It is therefore in the interest of the companies concerned to use these products to hedge their risk because of the sensitivity of their balance sheet to it. Moreover, the disruption of the seasons as well as other consequences of global warming such as the higher temperatures volatility could increase the demand for this type of product. This effect should even be accentuated given the growing impact of weather changes on our lives and our economy. We can take as an example the positive dependence between the occurrence of a pandemic and the increase of temperatures. The dramatic consequences of a pandemic, both economically and socially, are no longer in evidence.

All these reasons make it necessary to develop pricing and risk management methods for this type of product. Many pricing methods have been studied, especially since the 90's. Unfortunately, classical pricing methods based on the assumptions of market completeness, non-arbitrage and replicating strategy are not applicable in this case since this market is incomplete (Alexandridis and al (2013)). Indeed, it is not possible to buy, sell or store the underlying asset as is the case with a classical stock option.

We will therefore try in this thesis a different pricing approach based on neural networks and a discrete model for temperatures. We will first present all the elements necessary for a good understanding of the different steps in the construction of such models. We will also detail how these models learn from the data as well as the problems that can be encountered in this kind of task. Note that in this part we will not refer directly to temperatures since the goal is to explain how these models work regardless of the task.

Once all these elements are presented, it will be time to use all this knowledge in order to calibrate different models in order to be able to model reasonably the temperatures of Madrid. This will allow us to reconstruct the indexes and then to price different weather derivatives on these indexes using Monte Carlo simulations.

Chapter 1

Neural Networks

1.1 Neural Units

Before getting stared with entire neural networks, it is essential to understand how a single neuron works. Initially called "perceptron", it was introduced in 1957 by Frank Rosenblatt and was at the start a supervised learning algorithm for binary classification. In the context of supervised learning, the data is stored in n vectors $x_i = (x_{i1}, ..., x_{ip})^\intercal$ and the corresponding response y_i where $i \in \{1, 2, ..., n\}$ and where p is the number of explanatory variables. The explanatories variables can be either continuous or categorical but in this case, certain manipulations have to be made in order to transform them in a convenient form. The mostly used method is the introduction of dummy variables (cfr appendix 8.1). The observation x_i is weighted by p parameters $w_j \in \mathbb{R}$, usually called "weights" and one parameter $b \in \mathbb{R}$, usually called "bias". Then we can note that the weights vector $W \in \mathbb{R}^p$ and the bias vector $b \in \mathbb{R}$, which is a scalar in the case of a single neuron. All these parameters will be necessary to compute $z_i \in \mathbb{R}$, the output signal for observation vector x_i.

$$z_i = b + \sum_{j=1}^{p} \omega_j x_{ij} \tag{1.1.1}$$

Then, this signal passes through an activation function $g(.)$ to produce the ouput signal

$$\hat{y}_i = g\left(\sum_{j=1}^{p} \omega_j \times x_{ij} + b \right) \tag{1.1.2}$$

or in matrix form

$$\hat{y}_i = g\left(W^\intercal \bullet X + b \right) \tag{1.1.3}$$

In the inital model of Rosenblatt (1957), this activation function was the following

$$g(z) = \begin{cases} 1 & \text{si } \sum_{j=1}^{p} \omega_j \times x_{ij} > 0 \\ 0 & \text{sinon.} \end{cases} \tag{1.1.4}$$

We can see that the notion of bias did not exist in the initial model and that the activation function $g(.)$ was a step function. These are the two things that have been a problem for many years, preventing neural networks from expressing their full potential. Today, the "modern" perceptron has a bias and an activation function that has to be continuous, and strictly increasing. Usually, this is also lower and upper bounded, in order to keep the output signal in a defined range. This also allows to quantify at which degree a neuron is activated or not and to set a full activation limit. Note that $\hat{y}_i$ trying to estimate y_i. Note that $\hat{y}_i$ is a scalar given that the dimension of the dot product $W \bullet X$ equals to $(1 \times p)(p \times 1) = (1 \times 1)$. In the case where the activation function is the identity function, we are in the classical case of a linear regression, with all its limitations. In fact, this only allows to model linear dependencies between the explanatory variables X and the targets Y. This explains the interest of using non-linear function as activation function in order to model non linear dependencies.

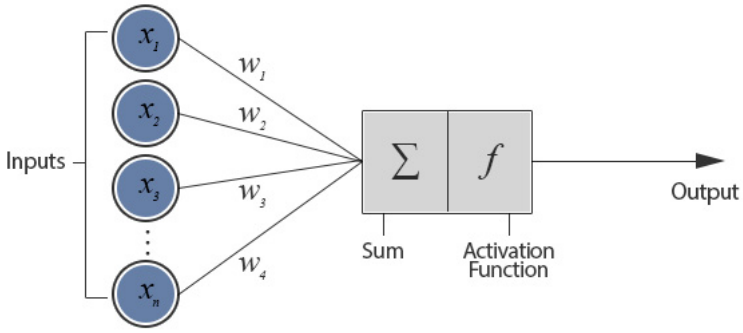


Figure 1.1.1: Neuron Unit

1.2 Activation function

In this section, we present the most common activation functions. However, it is easy to create them as long as they meet certain conditions mainly to be continuous, strictly increasing and at least one time differentiable.

1.2.1 Sigmoid function

The sigmoid function is defined as

$$f(x) = \frac{1}{1 + e^{-\lambda x}} \tag{1.2.1}$$

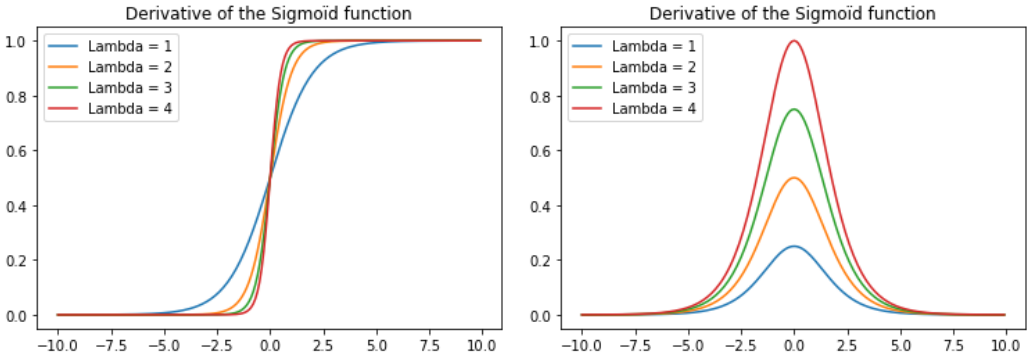
The function is
$\mathbb{R} \to [0, 1]$ defined. In fact, if we can calculate the limits we get

$$\begin{aligned} \lim_{x \to \infty} f(x) &= \frac{1}{1 + e^{-\infty}} = 1 \\ \lim_{x \to -\infty} f(x) &= \frac{1}{1 + e^{\infty}} = 0 \end{aligned} \tag{1.2.2}$$

The function is therefore upper and lower bounded with image ´$[0, 1]$. As referred in the previous section, this function approaches the binary behaviour of the initial step function used by Rosenblatt (1957) when λ increases but with the additional property of differentiability which is essential for the tuning process of the networks. This will be explain later in section 3.1. We can also compute the derivative of the sigmoid function.

$$\begin{aligned} \frac{df(x)}{dx} &= \frac{\lambda e^{-\lambda x}}{(1 + e^{-\lambda x})^2} \\ &= \lambda \left(\frac{1 + e^{-\lambda x} - 1}{(1 + e^{-\lambda x})^2} \right) \\ &= \lambda f(x)(1 - f(x)) \end{aligned}$$

We can observe that the derivative of this function can be written as a function of itself, which is interesting for computational calculation time. However, given that $f(x) \in [0, 1]$ we can highlight that when the neuron is activated strongly or very slightly, the derivative is almost zero. This detail will be important when calibrating a neural network. Usually in neural networks $\lambda = 1$.



(a) Sigmoid activation function (b) Derivative of the Sigmoid

Figure 1.2.1: Sigmoid function

Note that in the case of a single neuron, using the sigmoid ($\lambda = 1$) as activation function boils down to use a model of the same shape as a logistic regression. [1]

1.2.2 Hyperbolic tangeant function

The hyperbolic tangeant function is defined as

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.2.3}$$

The function is defined from $\mathbb{R} \to [-1, 1]$. In order to calculate the limits, we transform $f(x)$ as follows

$$\begin{aligned} f(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \times \frac{e^{-x}}{e^{-x}} \\ &= \frac{1 - e^{-2x}}{1 + e^{-2x}} \end{aligned}$$

We can calculate the limits

$$\begin{aligned} \lim_{x \to \infty} f(x) &= \lim_{x \to \infty} \frac{1 - e^{-2x}}{1 + e^{-2x}} \\ &= 1 \\ \lim_{x \to -\infty} f(x) &= \lim_{x \to -\infty} \frac{1 - e^{-2x}}{1 + e^{-2x}} \\ &= \lim_{x \to -\infty} \frac{-2e^{-2x}}{2e^{-2x}} \\ &= -1 \end{aligned} \tag{1.2.4}$$

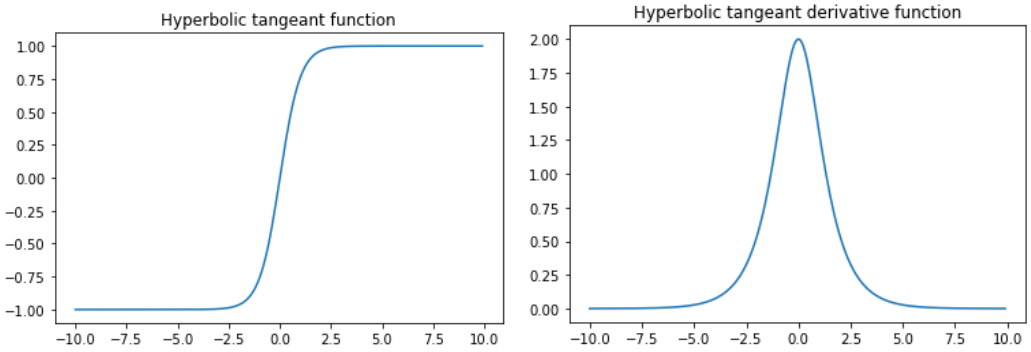
This function is also upper and lower bounded but the input is squashed on the interval $[-1, 1]$ instead of $[-1, 1]$ with the sigmoid function . Thanks to some trigonometric identities, we can compute the

[1] Note, however, that the method of estimating the various parameters is not necessarily identical, which is why we specify that only the shape is similar a priori.

$\tanh(x)$ derivative

$$\begin{aligned}\frac{d(\tanh(x))}{dx} &= \frac{d}{dx}\left(\frac{\sinh(x)}{\cosh(x)}\right)\\ &= \frac{d(\sinh(x))}{dx}\cosh(x)^{-1} + \sinh(x)\frac{d(\cosh(x)^{-1})}{dx}\\ &= \frac{\cosh^2(x) - \sinh^2(x)}{\cosh^2(x)}\\ &= \frac{1}{\cosh^2(x)}\\ &= sech^2(x)\end{aligned}$$

because $\cosh^2(x) - \sinh^2(x) = 1$.



(a) Hyperbolic tangent activation function (b) Derivative of the hyperbolic tangent

Figure 1.2.2: Hyperbolic tangent function

Compared to the sigmoid function, we observe that the derivative takes higher values which can have an important impact on the calibration. Its behavior is similar to that one the sigmoid when $\lambda = 2$ but with an image in the negative.

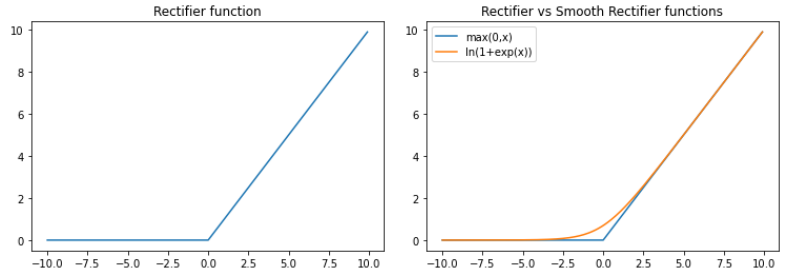
1.2.3 Relu function

In the machine learning framework, this function is defined as the positive part of its argument

$$f(x) = ax^+ = \max(0, ax) = a\max(0, x) \tag{1.2.5}$$

Unlike the first two activation functions, it is only lower bounded at zero, with no upper bound. The input is squashed on the interval $[0, \infty[$. Note that this function is differentiable everywhere except at zero which can cause a problem while calibrating the neural network. In order to solve this, we can arbitrarily fix the derivative at 0 as equal to 0 or 1. In this case, the limits are trivial $\lim_{x\to\infty} f(x) = \infty$ and $\lim_{x\to-\infty} f(x) = 0$. Note that its derivative on $\mathbb{R}_0^+$ is constant and equals a, usually fixed at $a = 1$. Note that there exists a smooth approximation of the rectifier function which is $f(x) = \ln(1 + e^x)$. This function, called "softplus", has the two following limits and derivative

$$\begin{aligned}&\lim_{x\to\infty} f(x) = \infty\\ &\lim_{x\to-\infty} f(x) = 0\\ &\frac{df(x)}{dx} = \frac{e^x}{1+e^x} = e^x e^{-f(x)}\end{aligned} \tag{1.2.6}$$



(a) Rectified linear unit activation function (b) Smooth rectified linear unit activation function

Figure 1.2.3: Rectified linear unit function

On the contrary of the rectifier function, this approximation are differentiable everywhere.
As Denuit and al.(2019) have pointed out, any cumulative distribution function defined on $\mathbb{R}$ are eligible candidates for activation function. They highlight the example of the normal distribution which, thanks to its thin tails, allows an activation close to 0 for scores outside the range $[-2, 2]$. This functions belongs to the radial basis function (RBF) which is a family of real-valued functions whose response depends only on the distance from the origin. So if we denote by $\phi(.)$ our RBF activation function, we can write

$$\phi(\omega_j x_{ij}) = \phi(||x_{ij}^{\omega}||_2) \tag{1.2.7}$$

where the ouput is a function of the Euclidean distance of x_{ij}^{ω} from the origin and where $x_{ij}\omega_j$ is the score of the i th individual.

It is essential to choose the activation function well because it will be directly used in the calibration process of the model as it will be detailed later. For the moment it is important to note that the properties of the derivative of the activation function used will have a direct impact on the quality of the convergence of the weights during the calibration of the model. Indeed, for example, the relu function will tend linearly towards 0 whereas the tanh function will tend in the manner of a negative exponential function. Moreover, the image of the activation function will directly determine the range of the output signal. Therefore, it is important to choose it judiciously on the last layer (cfr subsection 1.3.1) in order not to constrain our estimations range. For example, if we want to estimate a probability, it would be interesting to use a sigmoid activation function on the last neuron to ensure that the output is between 0 and 1. Therefore, it can be interpreted as a probability.

1.3 Neural Network architecture

We will limit ourselves here to present the most classical architectures. There are many of them, each one adapted to a specific type of problem (regression, text mining, image recognition, ...). Let us note however that the feed forward network is at the base of these different more exotic architectures and that it can be efficient in many of these different domains.

1.3.1 Feed forward neural network

Now that the basics about single neuron are clear, it will be easy to understand how a feed forward network works. A feed forward network is a type of neural network architecture where the information flows only in one direction, from the input layer up to the output layer, in opposition with the recurrent neural networks. Between these two ends, there is a certain number of hidden layers, each containing a certain number of neurons. All neurons of a certain hidden layer are connected to all the neurons of the layer that precedes it and the layer that follows it. Feed forward networks include the shallow networks (only 1 hidden layer) and the deep neural networks (more than 1 hidden layer). In deep

neural networks, the output of each neuron of the previous layer is used as input for the next layer, allowing to modelize complex relations between the input and the output.

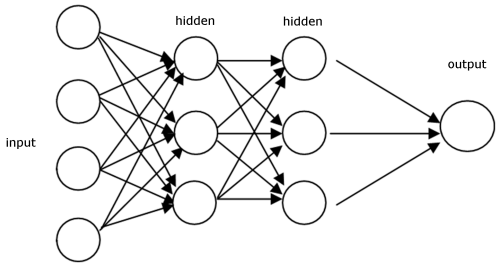
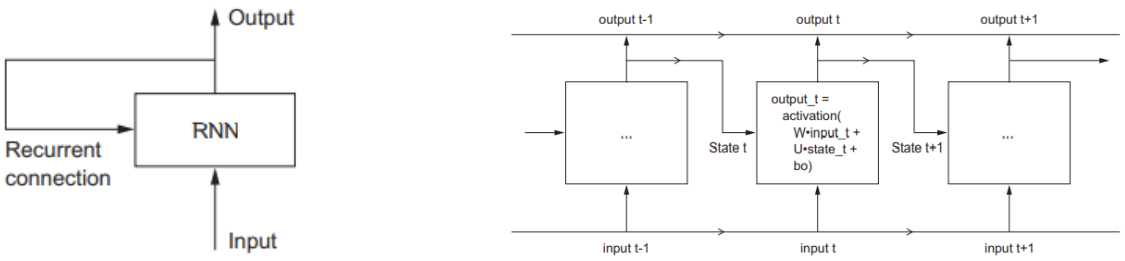


Figure 1.3.1: Feed forward neural network

1.3.2 Recurrent Neural Network

So far, we have presented the classical fully connected neural networks. These have the particularity of circulating information only in one direction and of not having memory. Indeed, each vector x_i is processed separately and no internal state of the network is stored between two consecutive input x_i. This is not necessarily optimal when processing a signal where the explanatory variables are arranged in a certain precise order in time. Let's take the example of a sentence where the position of each words has a capital importance. If we want to try to predict the next word of a sentence as a function of the previous words with a feed forward neural network, this implies transforming the explanatory variables, namely the previous words, into a single data point. However, it would be wise to have access to information about the order of arrival of the words and to be able to keep in memory the position of them in the sentence in order to predict more easily the next one. Chollet (2019) made an analogy with our human way of functioning. "When you read the sentence in this paragraph, you process it word by word, one eye movement at a time while keeping a memory of what was read before. In fact, you don't process the whole paragraph in one dense block. This sequential analysis allows a more fluid understanding of the meaning of the sentence. This is the idea of recurrent neural network (RNN)". A RNN adopts the same principle. Indeed, it processes each sequence element by element while keeping in memory a state of the information it has seen so far thanks to an internal loop that runs on each element of the sequence. This state is then reset between the processing of two independent sequences. Therefore, each sequence is represented as a unique data point but is processed element by element contrary to a feed forward neural network.



(a) RNN

(b) RNN unrolled

Figure 1.3.2: Recurrent neural net functionning

These architectures have what we call "memory cells" which are the building blocks of these RNN such as the neurons for feed forward neural networks. There are different types of memory cells (GRU,LSTM,...) but we will limit ourselves here to the presentation of the most powerful and known to date: the LSTM cells. This cell alows to fight a problem faced by previous RNN architecture called the "vanishing/exploding gradient". This resulted in stopping the learning of the model because of a poor control of the cell's memory.

1.3.2.1 LSTM units

LSTM has been introduced in 1995-1997 by S.Hochreiter and J.Schmidhuber and are now widely used by companies such as Google or Facebook. The recent researches are mainly conducted by Xu and al (2007).

A LSTM cell is composed of three gates: the forget gate, the input gate and the output gate. Each of them has a precise role allowing the cell to act on the current state of its memory. Note that in the three following sections, the index t will designate the iteration of the inner loop of this type of network as described above in section (1.3.2). The figure below will allow a better understanding of the functioning of the different gates.

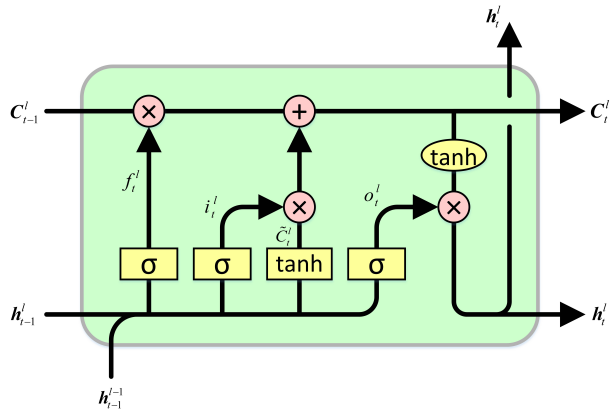


Figure 1.3.3: LSTM unit

1.3.2.1.1 Forget gate

As the name suggests, the objective of this gate is to select which information in the memory should be forgotten. In order to achieve this task, we apply the following transformation

$$f_t = \sigma(W_f^T[h_{t-1}, x_t] + b_f) \tag{1.3.1}$$

where $\sigma(.)$ is the sigmoïd function, $[h_{t-1}, x_t]$ corresponds to the concatenation of the vectors x_t and h_{t-1} and where W_f and b_f are new weights matrix. Note that we will present h_t after in the output gate explanation. We will denote the dimension of h_{t-1} as $(d \times 1)$ [2]. This implies that the dimension of $[h_{t-1}, x_t]$ is equal to $((d+p) \times 1)$ and the dimension of W_f^T and b_f are respectively equal to $(d \times (d+p))$ and $(d \times 1)$. More intuitively, we saw from section (1.2.1) that f_t will be a vector of dimension $(d \times 1)$ with values between zero and one. Then, this vector will be multiplied (elementwise multiplication) by C_{t-1}, the memory of the previous cell, computed thanks to the input gate that we will present just after. Intuitively, the information considered as useless and therefore to be forgotten in the memory corresponds to the values of f_t close to 0.

[2]The length of this vector can be chosen by the developer

1.3.2.1.2 Input Gate

Once the useless information is forgotten in the memory, we have to add new information provided by x_t and h_{t-1}. This operation is done with the two following operations

$$\begin{aligned} i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ \tilde{C}_t &= \tanh(W_C[h_{t-1}, x_t] + b_C) \end{aligned} \tag{1.3.2}$$

where we introduce new matrices W_i, W_C, b_i, b_C. Note that the dimensions of these new objects are the same as in the previous section. These operations have a double purposes. Firstly, $\tilde{C}_t$ brings new information given the state of the previous cell h_{t-1} and the new input x_t, with d values in $[-1, 1]$. Secondly, we have to filter this information in order to choose the right quantity of it to add to C_t, the memory of the cell. This is the role of i_t. In fact, given that its values are in $[0, 1]$, the elementwise multiplication between $\tilde{C}_t$ and i_t will fulfill this role. After these two operations, the memory vector C_t is updated by adding the new filtered information.

1.3.2.1.3 Output Gate

Once the state of the memory C_t is fixed, the state of the cell h_t must now be updated to be sent to the next LSTM cell. This is made through these new quantities

$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \times tanh(C_t) \end{aligned} \tag{1.3.3}$$

where we introduce new matrices W_o, b_o. Note that the dimensions of these new objects are the same as in the two last sections. The first vector is the ouput gate and consist again in d values between $[0, 1]$. Again the idea of o_t is to choose the right quantity of the memory in order to best characterizes the state of the cell. Note that we apply $\tanh(.)$ on C_t ton ensure that its values are in $[-1, 1]$ because it not necessary the case after adding new information to C_t.

1.4 Neural Networks calibration, main idea

Once the neural network is built, it must be calibrated. This implies the choice of a loss function $\mathcal{L}(.) : \mathbb{R}^2 \to \mathbb{R}$ which will compute the cost of estimating $\hat{y}_i$ instead of y_i. This function has to be continuous and accept a first order derivative. In some cases (cfr section 2.2), it is also necessary to accept a second derivative. It has to be chosen wisely because all the weights calibration will rely on it. In fact, the loss $\mathcal{L}(\hat{y}_i, y_i)$ is the signal that will be used to change all the weights and biases of the network in order to find the best combination that minimizes $\mathcal{L}$. If we denote all the weights and biases by Ω, the problem is

$$\Omega^* = \underset{\Omega}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y) \tag{1.4.1}$$

Therefore it is essential to choose a function that describes well the distance between the estimation $\hat{y}_i$ ans the true value y or more generally, a loss that describes well the objective of the problem at hand. Note that once the algorithm is launched, it will try to find any shortcut to minimize the loss function. Therefore, the loss has to be a good metric to quantify the success of the task at hand. In their researches, Denuit and al. (2019) advise to use the notion of deviance as loss function. The deviance is defined as follows

$$D^*(y_i, \hat{y}_i) = 2(l(y_i) - l(\hat{y}_i)) \tag{1.4.2}$$

where $l(.)$ stands for the log-likelihood function. If we consider a series of random variables X_i i.i.d. of

density $f_X(\Theta)$ with $i \in \{1, ..., n\}$ and the parameters vector Θ, we define the log likelihood function as

$$\begin{aligned} l_X &= \ln \prod_{i=1}^{n} f_{X_i}(\theta) \\ &= \sum_{i=1}^{n} \ln f_{X_i}(\theta) \end{aligned} \tag{1.4.3}$$

The idea is that, given all our observations x_i, which can be seen as one realisation of X_i, what is the value of $\Theta = (\theta_1, ..., \theta_m)$ that maximizes the probability to observe these data. For instance, if we assume that $f_{X_i}(\Theta) \sim \mathcal{N}(\mu_i, \sigma_i^2)$, m = 2. The deviance allows to take into account the statistical properties of y and allows to compare the goodness of fit of several models. Note the special case where $\hat{y} = y$, called the saturated model. This model perfectly fits the data so its log-likelihood is the best we can obtain and its deviance the lowest.

Chapter 2

Minimization algorithm

2.1 Gradient descent and Newton method

2.1.1 Gradient descent

A common way to minimze a function is to use a gradient descent. This is the simplest method to achieve this objective. However, note that this method shows some limitations that will be explained in the following sections. The main idea is to take a step iteratively in the opposite direction of the gradient of the loss function. The gradient of a function $f(x_1, ..., x_n)$ is a vector denoted by $\nabla f(x_1, ..., x_k) \in \mathbb{R}^k$ that contains $\frac{\partial f(x_1,...,x_k)}{x_i} \quad \forall i \in \{1, ..., k\}$. In the case of neural networks, if we consider that we have n parameters ω_i[1] to calibrate, it consists in choosing the new weight vector $\Omega_t \in \mathbb{R}^n$ as follows

$$\Omega_{t+1} = \Omega_t - \gamma \nabla \mathcal{L}(\Omega_t) \tag{2.1.1}$$

where $\nabla \mathcal{L}(\Omega_t)$ is the gradient vector of the loss function $\mathcal{L}$ evaluated at Ω_t, so a vector with n components, each corresponding to $\frac{\partial \mathcal{L}}{\partial \omega_i}$. Note that it works because $\mathcal{L}(\Omega_{t+1}) < \mathcal{L}(\Omega_t)$. But is it always the case? We can approximate $\mathcal{L}(\Omega_{t+1})$ by its first order Taylor's expansion which gives

$$\begin{aligned} \mathcal{L}(\Omega_{t+1}) &\approx \mathcal{L}(\Omega_t) + \nabla \mathcal{L}(\Omega_t)\ (\Omega_{t+1} - \Omega_t) \\ &\approx \mathcal{L}(\Omega_t) + \nabla \mathcal{L}(\Omega_t)\ (-\gamma \nabla \mathcal{L}(\Omega_t)) \end{aligned} \tag{2.1.2}$$

thanks to (2.1.1). This implies that

$$\mathcal{L}(\Omega_{t+1}) - \mathcal{L}(\Omega_t) = -\gamma (\nabla \mathcal{L}(\Omega_t))^2 < 0 \tag{2.1.3}$$

If we consider that $\mathcal{L}$ has a bowl shape, we can represent the progression of this method by the figure (2.2.1a). The red arrow shows the direction of the negative gradient at its origin. Note that the gradient at a point is orthogonal to the contour line going through that point. As we can observe, the method leads us to the bottom of the bowl. However it is not likely that the loss $\mathcal{L}$ is convex with a global minimum. Therefore, it is recommended to repeat the whole process with different starting points to escape local minimums even if it does not guarantee to find the absolute minimum if it exists.

Note that in a supervised learning context with n observation vectors $x_i \in \mathbb{R}^p$ and their corresponding target y_i, the gradient used corresponds to $\frac{1}{n} \sum_{i=1}^{n} \nabla \mathcal{L}(x_i, y_i, \Omega_t)$. This is the "true" gradient of the problem. Then this requires a lot of computations by iteration, exactly $n \times p$ calculations for each updates of Ω_t. However, other techniques are developed to improve the calculation time (cfr section (2.4)).

[1] In order to facilitate the writings and the understanding, we consider here that Ω_t contains the full set of weights w and bias b at the same time for iteration t.

2.2 Newton method

As the name of this method indicates, it was developed by I.Newton and J.Raphson in the 17-18th century before being considerably generalize by Thomas Simpson during the 18th century.

From equation (2.1.1) we can ask ourselves what is the optimal γ step size to choose in order to minimize $\mathcal{L}(\Omega_{t+1})$. We can rearrange equation (2.1.1) as

$$\Omega_{t+1} - \Omega_t = -\gamma \nabla \mathcal{L}(\Omega_t) \tag{2.2.1}$$

For ease of notation, let us assume that $z = \Omega_{t+1} - \Omega_t$. Thanks to the Taylor's expansion, we can write

$$\mathcal{L}(\Omega_{t+1}) = \mathcal{L}(\Omega_t) + \nabla \mathcal{L}(\Omega_t)\ z + \frac{1}{2} H(\Omega_t)\ z^2 + \mathcal{O}(3) \tag{2.2.2}$$

Where $H(\Omega_t)$ is the Hessian matrix of $\mathcal{L}(\Omega_t)$. To continue the previous example, the Hessian matrix of a function $f(x_1, ..., x_k)$ is a matrix $H_{f(x_1,...,x_k)} \in \mathbb{R}^{k \times k}$ that contains $\frac{\partial f(x_1,...,x_k)}{\partial x_i \partial x_j}\ \ \forall i, j \in \{1, ..., k\}$.

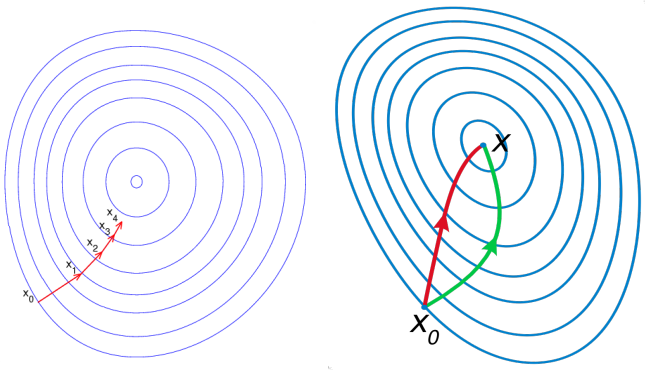
The second order Taylor's expansion of $\mathcal{L}(\Omega_{t+1})$ is a quadratic function in z. Then, we know that we have to impose $\frac{d\mathcal{L}(\Omega_{t+1})}{dz} = 0$ in order to find the combination Ω^* that minimizes [2] our loss.

$$\frac{d\mathcal{L}(\Omega_{t+1})}{dz} = \nabla \mathcal{L}(\Omega_t) + H(\Omega_t)\ z = 0 \tag{2.2.3}$$

From equation (2.2.3) we can extract the optimal solution which is $z = -H(\Omega_t)^{-1} \nabla \mathcal{L}(\Omega_t)$. By substitution in equation (2.2.1), we compute the optimal step size $\gamma = H(\Omega_t)^{-1}$. This solutions allows us to update equation (2.1.1) as follows

$$\Omega_{t+1} = \Omega_t - H(\Omega_t)^{-1} \nabla \mathcal{L}(\Omega_t) \tag{2.2.4}$$

Again, if we consider that $\mathcal{L}$ has a bowl shape, we can represent the progression of this method by the figure (2.2.1b). Newton's method takes curvature into account to take a more direct route. Therefore, it is faster but can lead to zigzagging. Moreover, if the initial value is far from the true zero, it can fail to converge.



(a) Gradient descent (b) Newton descent

Figure 2.2.1: Gradient descent vs Newton Method

Unfortunately, nothing ensures that our function is convex and it is likely that $\mathcal{L}$ has local minimums. This implies to be careful not to get stuck in a local minimum. Moreover, from a computational time, $H(\mathcal{L}(\Omega_t))$ is often of high dimension and therefore the calculation of its inverse is time consuming and

[2] Here, we assume that the loss function has a perfect convex bowl shape in order to be sure that we minimize. This implies that the hessian matrix is positive semi-definite

it assumes that $H(\mathcal{L}(\Omega_t))$ is well conditioned, which is not always the case. Note that other methods based on gradient descent were developed in order to solve the weaknesses of the previous methods. In the following sections, we will present how this simple gradient descents can evolve to become more robust.

2.3 Gradient descent extensions

2.3.1 Momentum effect

As B.Wang and al. (2020) explained in their researches that a first improvement was the addition of momentum. This aims at increasing the speed of convergence and to smooth the oscillations of the loss during training. An analogy is often made between this algorithm and the image that we would have of a ball going down a slope more and more quickly in the right direction, that is to say towards the minimum. Indeed, the faster it goes, the harder it is to change its trajectory.

Algorithm 1 Gradient Descent Algorithm with momentum

Initialization

- Randomly choose the values of Ω_0

for t=0 to maximum epoch T **do**
  Calculate $v_{t+1} = \eta v_t + \gamma \nabla \mathcal{L}(\Omega_t)$ ▷ Momentum update
  Update the weights $\Omega_{t+1} = \Omega_t - v_{t+1}$
end for

Note that v_t is the update vector of the past time step. If we observe the effect of momentum a little better, we can develop the expression of v_{t+1}

$$\begin{aligned} v_{t+1} &= \eta v_t + \gamma \nabla \mathcal{L}(\Omega_t) \\ &= \eta \left(\eta v_{t-1} + \gamma \nabla \mathcal{L}(\Omega_{t-1}) \right) + \gamma \nabla \mathcal{L}(\Omega_t) \\ &= \eta \left(\eta \Big(\eta v_{t-2} + \gamma \nabla \mathcal{L}(\Omega_{t-2}) \Big) + \gamma \nabla \mathcal{L}(\Omega_{t-1}) \right) + \gamma \nabla \mathcal{L}(\Omega_t) \\ &= \eta^{n+1} v_{t-n} + \gamma \Big[\nabla \mathcal{L}(\Omega_t) + \eta \nabla \mathcal{L}(\Omega_{t-1}) + ... + \eta^n \nabla \mathcal{L}(\Omega_{t-n}) \Big] \\ &= \eta^{n+1} v_{t-n} + \gamma \sum_{i=0}^{n} \eta^i \nabla \mathcal{L}(\Omega_{t-i}) \end{aligned} \tag{2.3.1}$$

with $0 < \eta < 1$. We observe that at time t, the influence of the momentum computed n times before is exponentially decreasing. We also observe that the momentum term increases for dimensions whose gradients point in the same direction and reduces for dimension whose gradient change of directions between consecutive iteration. This means faster convergence and reduced oscillations - recall the analogy. Unfortunately, this leads to a new problem put forward by Yuri Nesterov (1983). Once close to the target, the momentum is often very high and should therefore decrease in order to avoid going completely through the target. He therefore proposed to modify the previous algorithm as follows.

2.3.2 Nesterov Accelerated Gradient Algorithm

Algorithm 2 Nesterov Accelerated Gradient Algorithm

1: **Initialization**
   - Randomly choose the values of Ω_0

2: **for** t=0 to maximum epoch T **do**
3: Calculate $v_{t+1} = \eta v_t + \gamma \nabla \mathcal{L}(\Omega_t - \eta v_t)$ ▷ New momentum update
4: Update the weights $\Omega_{t+1} = \Omega_t - v_{t+1}$
5: **end for**

Unlike the previous algorithm, we first make a jump based on the previous momentum before computing the gradient. As explained by Chi and al. (2017), this can be seen as a correction factor to account for the previous momentum accumulation. This more anticipatory update prevents us from going too fast and we are more responsive to changes. The idea of more dynamic learning, of adapting our updates to the slope of the error function is a good one. This idea can be extended to the learning rate leading to AdaGrad Algorithm proposed by Duchi and al. (2011). In this method, the learning rate can adapt based on the parameters.

2.3.3 Adagrad

Algorithm 3 AdaGrad Algorithm

1: **Initialization**
   - Randomly choose the values of Ω_0

2: **for** t=0 to maximum epoch T **do**
3: Calculate $A_t = \sum_{j=0}^{t} \nabla \mathcal{L}(\Omega_j)^\intercal \nabla \mathcal{L}(\Omega_j)$
4: Compute $a_t = diag(A_t)$
5: Update the weights $\Omega_{t+1} = \Omega_t - \gamma(a_t + \epsilon)^{-\frac{1}{2}} \otimes \nabla \mathcal{L}(\Omega_t)$ ▷ Where $\otimes$ is the elementwise product.
6: **end for**

We observe that we divide the previous learning rate γ by the square root of the vector $a_t \in \mathbb{R}^n$ if we consider that we have n parameters to calibrate. This vector a_t contains the $L2$ norms of all the previous gradients. Then, we obtain a vector of n learning rates, one for each of the n parameters of Ω_t. We can add that each component of a_t is an increasing function of the iteration t. This causes a decrease in the learning rates, allowing it to be self-managing for each of the elements of Ω. This allows to facilitate the convergence of the algorithm. In fact, if one parameter has been changed a lot during the previous iteration, it will be changed less in the next step given the accumulation of the gradients in the a_t vector. However, one problem that we possibly encounter with this algorithm is that certain components of a_t become too big. In fact, this will strongly decrease the associated learning rates until it stop learning in the corresponding directions. A possible solution to this is to restrict the window over which the gradient accumulation is considered and to take an exponentially decreasing importance with time of the previous gradients. This allows not to accumulate all past gradients. These adaptations make up the Adadelta algorithm. Unlike Adagrad, the latter will continue to learn even after many updates. A final possible improvement is the addition of adaptive momentum. This addition is the basic idea of the Adam [3] and the RMSprop[4] algorithms. We will first begin to present the latter because it is slightly simpler than Adam.

Note that $\epsilon > 0$ but is very small. This allows to respect the domain of $f(x) = \sqrt{x}$ which is defined on $\mathbb{R}_0^+$. In a few words, it prevents to divide by zero.

[3] Adam stands for Adaptative moment estimation
[4] RMSprop stands for Root Mean Squared Propagation

2.3.4 RMSprop

Algorithm 4 RMSprop Algorithm

1: **Initialization**
- Randomly choose the values of Ω_0
- $b_0 \leftarrow 0$

2: **for** t=0 to maximum epoch T **do**
3: Calculate $b_{t+1} = \beta b_t + (1-\beta)\nabla\mathcal{L}(\Omega_t)^2$
4: Update the weights $\Omega_{t+1} = \Omega_t - \frac{\gamma}{\sqrt{b_t+\epsilon}}\nabla\mathcal{L}(\Omega_t)$
5: **end for**

Note that here, $\nabla\mathcal{L}(\Omega_t)^2$ is a vector containing the square of each of its components, it is not a matrix product so it keeps the same dimension. Here, as for Adagrad, we adapt the learning rate individually for each element of Ω_t but this time taking into account the square root of a momentum function. The momentum rate β, or in other words the forget rate of the past computations, appears in the computation of the vector b_t which is a moving average. This allows to prevent the problem encountered by Adagrad where the learning rate was vanishing to zero through the epochs because of the accumulation of all the past gradients. Note that this algorithm has been developed at Toronto by Hinton during one his lesson but has not been published in a formal academic paper.

2.3.5 Adam

Once RMSprop is understood, it is not complicated to move to Adam. As explained by Denuit. and al (2019), this algorithm also considers the first moment but differs from the previous algorithm by also taking into account the second moment in the update of the weights of the gradient descent. To be as clear as possible, we first present the algorithm as a whole before going into the interpretation of its equations.

Algorithm 5 Adam Algorithm

1: **Initialization**
- Randomly choose the values of Ω_0
- $m_0 \leftarrow 0$
- $v_0 \leftarrow 0$

2: **for** t=0 to maximum epoch T **do**
3: Calculate
- $m_{t+1} = \beta_1 m_t + (1-\beta_1)\nabla\mathcal{L}(\Omega_t)$
- $v_{t+1} = \beta_2 v_t + (1-\beta_2)\nabla\mathcal{L}(\Omega_t)^2$

4: Compute
- $\hat{m_{t+1}} = \frac{m_{t+1}}{1-\beta_1^t}$ ▷ First moment bias corrected term
- $\hat{v_{t+1}} = \frac{v_{t+1}}{1-\beta_2^t}$ ▷ Second moment bias corrected term

5: Update the weights $\Omega_{t+1} = \Omega_t - \frac{\gamma}{\sqrt{\hat{v}_t+\epsilon}}\hat{m}_t$
6: **end for**

Adams keeps tracking the two first moments of the last gradients through moving averages. This is represented by the m_t and v_t terms. Therefore we can interpret β_1 and β_2 as the speeds at which the algorithm forgets the previous computations. Note that it was found empirically that $\beta_1 = 0.9$ and $\beta_2 = 0.999$ show good performance. After rescaling, we update the basis constant learning rate γ thanks to $\hat{m}_t$ and $\hat{v}_t$.

2.4 Gradient update frequency

After presenting and understanding each of the above algorithms, there are still two important questions to ask. First, is there an optimal frequency for gradients computation? How to efficiently calculate numerically the gradient that appears in each of the above algorithms?

To answer the first question, we can turn the question as follows: "Is it better to calculate the gradient once all the data have been seen by the model, so the "true gradient" or to calculate it for each data individually? The answer to this question is as often a story of trade-offs. This will be the object of the two following sections.

2.4.1 Stochastic Gradient Descent

This algorithm update the weights for each data that passes through the network. These data are picked up randomly. In pseudo-code, we can note

Algorithm 6 Stochastic Gradient Descent Algorithm

Initialization Randomly choose the values of Ω_0
for t=0 to maximum epoch T **do**
    Shuffle(data)
    for i in range(iterations) **do** ▷ Consider that we draw "iterations" random data
        Calculate $\nabla\mathcal{L}(\Omega_{t,i})$
        Update the weights $\Omega_{t,i+1} = \Omega_{t,i} - \gamma\nabla\mathcal{L}(\Omega_{t,i})$
    end for
end for

In other words, this means that if we have n data we update the gradient n times per epoch [5]. This has the effect of causing larger fluctuations, so a higher variance, in the loss function because of the high update frequency. Indeed, if two consecutive input are very different, for example if one of them is an outlier, it will cause a jump in the update of the weights. This has an advantage since it allows the algorithm to decrease the chances of getting stuck in a local minimum and drastically decrease computational time. On the contrary, the classical gradient descent will converge to the nearest local minimum, hence the importance of initializing the algorithm with several starting points in this case. Unfortunately, the SGD may fail to converge because of its high update frequency. Contrary to what was put forward in section (2.1.1), Bottou (2012) explains taht the following simplification operation is performed $\nabla\mathcal{L}(X, \Omega_t) \approx \nabla\mathcal{L}(x_i, \Omega_t)$, if X corresponds to the $(n \times p)$ matrix containing all the x_i vectors in rows. Therefore, since the data are randomly drawn from the empirical distribution, it is expected that this approximation behaves as its expectation, so the true gradient. Then the SGD performance depends on the examples that are drawn, in other words the stochastic process $\{\Omega_t, t = 1, ...\}$ depends on the data drawn.

[5] One epoch represents the moment when all the data is passed through the network once

2.4.2 Mini-batch Gradient Descent

This method is a compromise between a too fast or too slow update frequency. It allows to take advantage of both and has become a standard in machine learning. In this case, the algorithm performs an update for each subset of training examples that we can decide the size of.

Algorithm 7 Mini-batch Stochastic Gradient Descent Algorithm

Initialization Randomly choose the values of Ω_0
for t=0 to maximum epoch T **do**
  Shuffle(data)
  for batch in get-batches(data,batchsize = n) **do** ▷ This loop contains $\frac{nrow(data)}{batchsize}$ steps
    Calculate $\nabla\mathcal{L}(\Omega_{t,batch})$
    Update the weights $\Omega_{t,batch+1} = \Omega_{t,batch} - \gamma\nabla\mathcal{L}(\Omega_{t,batch})$
  end for
end for

In order to be clear, here is a simple example. Assume we have 10.000 training data. For each of the three previous methods, you can find the number of weights update per epoch in the table below.

Gradient Descent	1
Stochastic Gradient Descent	10.000
Mini-batch Stochastic Gradient Descent (n=50)	200

Table 2.4.1: Number of gradient update per epoch

Note that all the previous algorithms (Adagrad,Adam,RMSprop,...) can work with mini-batches and are all implemented in keras [6]. Zinkevich and al. (2010) specifies what modifications need to be made to the various algorithms to allow them to be used with batches.

Now, we have to answer the second question which was: "How to efficiently tune the weights in the context of a feed forward network with multiple layers?". A smart way to achieve this task in a neural network is the backpropagation algorithm that we present in the next chapter.

[6]Keras is a free deep learning API developped by Google engineers, running on top of the machine learning platform Tensorflow. It allows to easily built simple but also complex neural networks architecture for machine learning application.

Chapter 3

Backpropagation Algorithm

Before going into the derivation of the equations of the backpropagation algorithm, it is important to understand its essence. The objective of this algorithm is to formulate an efficient way to change the weights of the neural network. As this model can be seen as a compound function (cfr section 1.1) as explain Chollet and al. (2018), the chain derivative rule will play a crucial role in the calculations. As already mentioned, the information flow in the network can be represented by the following figure

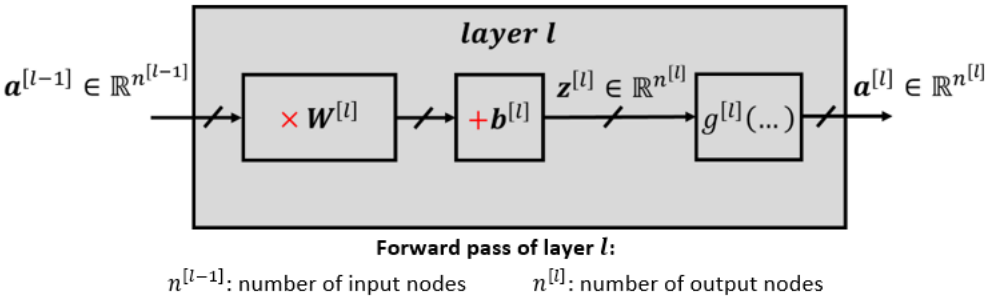


Figure 3.0.1: Feed forward information flaw

Where $l \in \{1, 2, ..., L\}$ indicates the layer, $W^{[l]}$ and $b^{[l]}$ are respectively the weight matrix and the bias vector of layer l and where $z^{[l]}$ is the score vector of layer l before being transform through $g(.)^{[l]}$, the activation function of layer l, into $a^{[l]}$ the final signal of layer l. Note that $a^{[l]}$ will be the input signal of layer $l+1$, therefore we can notice that $z^{[l+1]}$ is a function of $a^{[l]}$. If we denote the number of neurons of a particular layer l by $n^{[l]}$, we can notice that the layer l can be seen as a function that is $\mathbb{R}^{n^{[l-1]}} \to \mathbb{R}^{n^{[l]}}$ defined thanks to the fact that $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$. The idea of backpropagation algorithm is to reverse this process by sending a back signal of the model error from the last layer L to the $L-1$ layer. Then, this signal goes from the layer $L-1$ to the layer $L-2$ and so on until the first layer. Given that this back signal depends on the error, the weights of each neurons will be update thanks to this information.

3.1 Backpropagation equation derivation

This section draws on the work of Kurbiel (2021). The backpropagation algorithm starts in the layer L, so the last layer, and successively moves back one layer at a time. For each visited layer, we compute the layer error $\frac{\partial \mathcal{L}}{\partial z^L}, \frac{\partial \mathcal{L}}{\partial z^{L-1}}, ...$ If we assume that we arrive at layer l, we can rewrite our loss function $\mathcal{L}$ as a function of the weighted input z^l of this particular layer. Moreover, given the relation between z^l and a^{l-1}, we can write our loss function $\mathcal{L}$ as a function of a^{l-1}.

$$\mathcal{L}(z_1^{[l]}, z_2^{[l]}, ..., z_{n^l}^{[l]}) = \mathcal{L}\left(z_1^l(a_1^{[l-1]}, ..., a_{n^{[l-1]}}^{[l-1]}), z_2^l(a_1^{[l-1]}, ..., a_{n^{[l-1]}}^{[l-1]}), ... , z_{n^l}^l(a_1^{[l-1]}, ..., a_{n^{[l-1]}}^{[l-1]})\right) \tag{3.1.1}$$

Note that we can compute the value of a particular z_k^l with the following relations

$$z_k^l = w_{k,1}^{[l]}\ a_1^{[l-1]} + ... + w_{n^{[l-1]},1}^{[l]}\ \ a_{n^{[l-1]}}^{[l-1]}\ \forall k \in \{1, ..., n^{[l]}\} \tag{3.1.2}$$

Therefore, thanks to the chain rule we are now able to compute

$$\frac{\partial \mathcal{L}(a_1^{L-1}, ..., a_{n^{l-1}}^{l-1})}{\partial a_i^{l-1}} = \sum_{k=1}^{n^{[l]}} \frac{\partial \mathcal{L}(z_1^{[l]}, ..., z_{n^l}^{[l]})}{\partial z_k^l}\ \frac{\partial z_k^{[l]}(a_1^{[l-1]}, ..., a_{n^{l-1}}^{[l-1]})}{\partial a_i^{[l-1]}} \qquad \forall i \in \{1, ..., n^{[l-1]}\} \tag{3.1.3}$$

The first term of (3.1.3) is the error of layer l and has already been computed in the previous step of the backpropagation. The second term is easy to compute because it equals zero except when computing $w_{k,i}\frac{\partial a_i^{[l-1]}}{\partial a_i^{[l-1]}} = w_{k,i}^{[l]}$. This remark allows to simplify (3.1.3)

$$\frac{\partial \mathcal{L}}{\partial a_i^{l-1}} = \sum_{k=1}^{n^{[l]}} \frac{\partial \mathcal{L}(z_1^{[l]}, ..., z_{n^l}^{[l]})}{\partial z_k^l}\ w_{k,i} \qquad \forall i \in \{1, ..., n^{[l-1]}\} \tag{3.1.4}$$

or in the matrix form

$$\frac{\partial \mathcal{L}}{\partial a^{l-1}} = \left[W^{[l]}\right]^\intercal \frac{\partial \mathcal{L}}{\partial z^{[l]}} \tag{3.1.5}$$

Where $W^{[l]} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}$, $\frac{\partial \mathcal{L}}{z^{[l]}} \in \mathbb{R}^{n^{[l]}}$ and where $\frac{\partial \mathcal{L}}{a^{[l-1]}} \in \mathbb{R}^{n^{[l-1]}}$. At this point, we have propagated the error of layer l through the weights and biases and we have arrived at layer $l-1$. In order to get the error of the $l-1$ layer, we have to backpropagate through the activation function of the $l-1$ layer thanks this relation

$$a_i^{[l-1]} = g^{[l-1]}(z_i^{[l-1]}) \quad \forall i \in \{1, ..., n^{[l-1]}\} \tag{3.1.6}$$

Therefore, applying the chain rule we obtain

$$\frac{\partial \mathcal{L}(z_1^{[l-1]}, ..., z_{n^{l-1}}^{[l-1]})}{\partial z_i^{[l-1]}} = \sum_{k=1}^{n^{[l-1]}} \frac{\partial \mathcal{L}(a_1^{[l-1]}, ..., a_{n^{[l-1]}}^{[l-1]})}{\partial a_k^{[l-1]}}\ \frac{\partial g^{[l-1]}(z_k^{[l-1]})}{\partial z_i^{[l-1]}} \tag{3.1.7}$$

We can observe that the first term in (3.1.7) is the one that we already computed in (3.1.3). The second term can be computed easily since the activation function $g(.)$ takes only one z as input

$$\frac{\partial g^{[l-1]}\left(z_k^{[l-1]}\right)}{\partial z_i^{[l-1]}} = \begin{cases} g^{[l-1]\prime}\left(z_k^{[l-1]}\right) & \text{if } k = i \\ 0 & \text{else} \end{cases} \tag{3.1.8}$$

We can see in
It allows to understand the previous remarks made in section 1.2. Then, we can rewrite (3.1.7) as follows

$$\frac{\partial \mathcal{L}}{\partial z_i^{[l-1]}} = \left[W^{[l]}\right]^T \frac{\partial \mathcal{L}}{\partial z^{[l]}} * g^{[l-1]\prime}\left(z^{[l-1]}\right) \tag{3.1.9}$$

by substituting (3.1.5) and (3.1.8) in (3.1.7), where $*$ stands for the elementwise product. Since z depends directly on the weights and biases, this last result will allow us to establish the sensitivity of the loss to the different parameters of Ω in a relatively direct way. If we start with the weights, we can write

$$\frac{\partial \mathcal{L}(w_{1,1}^{[l]}, ..., w_{n^{[l]},n^{[l-1]}}^{[l]})}{\partial w_{i,j}^{[l]}} = \sum_{k=1}^{n^{[l]}} \frac{\partial \mathcal{L}(z_1^{[l]}, ..., z_{n^{[l]}}^{[l]})}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}(w_{1,1}, ..., w_{n^{[l]},n^{[l-1]}})}{\partial w_{i,j}^{[l]}} \tag{3.1.10}$$

We recognize again the first term which is nothing else than the error of the layer. Again, the second term is easy to calculate and gives

$$\frac{\partial z_k^{[l]}}{\partial w_{i,j}^{[l]}} = \begin{cases} a_j^{[l-1]} & \text{if } k = i \\ 0 & \text{else} \end{cases} \tag{3.1.11}$$

This is due to the fact that each element of z only depends on a single row of the weight matrix. In fact, if we would like to compute z_k, we obtain a solution that only depends on the row k of the weight matrix.

$$z_k^{[l]} = w_{k,1} a_1^{[l-1]} + w_{k,2} a_2^{[l-1]} + ... + w_{k,,n^{[l-1]}} a_{n^{[l-1]}}^{[l-1]} + b_k^{[l]} \quad \forall k \in \{1, ..., n^{[l]}\} \tag{3.1.12}$$

At the end, we obtain the expression of the weights gradient which is

$$\frac{\partial \mathcal{L}(w_{1,1}^{[l]}, ..., w_{n^{[l]},n^{[l-1]}}^{[l]})}{\partial w_{i,j}^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_i^{[l]}} a_j^{[l-1]} \tag{3.1.13}$$

or in a matrix form

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{\partial \mathcal{L}}{\partial z^{[l]}} [a^{[l-1]}]^T \tag{3.1.14}$$

with

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} \in \mathbb{R}^{n^{[l]} \times n^{[l-1]}}, \ \frac{\partial \mathcal{L}}{\partial z^{[l]}} \in \mathbb{R}^{n^{[l]}}, \ a^{[l-1]} \in \mathbb{R}^{n^{[l-1]}} \tag{3.1.15}$$

All that remains is to determine the sensitivity of the loss to bias. Following the same reasoning as for the weights we can write

$$\frac{\partial \mathcal{L}(b_1^{[l]}, ..., b_{n^{[l]}}^{[l]})}{\partial b_i^{[l]}} = \sum_{k=1}^{n^{[l]}} \frac{\partial \mathcal{L}(z_1^{[l]}, ..., z_{n^{[l]}}^{[l]})}{\partial z_k^{[l]}} \frac{\partial z_k^{[l]}(b_1^{[l]}, ..., b_{n^{[l]}}^{[l]}))}{\partial b_i^{[l]}} \tag{3.1.16}$$

Again, the first term is already known and the second is trivial

$$\frac{\partial z_k^{[l]}}{\partial b_i^{[l]}} = \begin{cases} 1 & \text{if } k = i \\ 0 & \text{else} \end{cases} \tag{3.1.17}$$

This is again related to the fact that each element of z depends on only one b as we can see in (3.1.12).We finally obtain

$$\frac{\partial \mathcal{L}(b_1^{[l]}, ..., b_{n^{[l]}}^{[l]})}{\partial b_i^{[l]}} = \frac{\partial \mathcal{L}}{\partial z_i^{[l]}} \tag{3.1.18}$$

or in matrix form

$$\frac{\partial \mathcal{L}}{\partial b^{[l-1]}} = \frac{\partial \mathcal{L}}{\partial z^{[l-1]}} \tag{3.1.19}$$

with

$$\frac{\partial \mathcal{L}}{\partial b^{[l-1]}} \in \mathbb{R}^{n^{[l-1]}}, \ \frac{\partial \mathcal{L}}{\partial z^{[l-1]}} \in \mathbb{R}^{n^{[l-1]}} \tag{3.1.20}$$

This now allows us to calculate $\nabla \mathcal{L}$ efficiently for all elements of Ω which contain all the weights and the biases for the ease of notation. This makes it possible to use all the gradient descent algorithms previously presented. Moreover, it is possible to adapt the backpropagation to a stochastic gradient descent method with mini-batch thanks to an averaging system.

Chapter 4

Neural Network Finetuning

4.1 Hyper parameters optimization algorithm

The methods presented in the previous sections allow us to calibrate the weights of the neural network thanks to the gradient descent and the backpropagation. However, other parameters have a considerable impact on the performance of the model such as, for example, the number of layers and the number of neurons per layer which fixes the hypothesis space in which the network can evolve. These are called "Hyperparameters". It is not possible to estimate them from the data. This is partially due to the fact that it is not possible to use gradient techniques to improve the architecture of the neural network since the loss function does not take into account these hyperparameters. For example, the derivative of the loss with respect to the number of layers has no meaning because of the discrete domain of this quantity and because it is not possible to build an analytic function of the loss in function of the number of layers. Hyperparameters tuning has been left to humans for a long time, which has fed the idea that machine learning is "more of an art than a science"; as Chollet and al. (2018) say. However, some techniques have been developed to automate this process. As before, the problem to solve is

$$\Omega^* = \underset{\Omega}{\operatorname{argmin}} \sum_{i=1}^{n} \mathcal{L}(\hat{y}_i, y) \tag{4.1.1}$$

where Ω is the hyperparameters vector and
$\mathcal{L}(\hat{y}_i, y_i)$ is the loss function that gives the cost of estimating $\hat{y}$ instead of y . We will therefore present in the following three sections different methods used to perform this task.

4.1.1 Grid Search

Assuming n hyperparameters, this method consists in defining a discrete space with n dimensions. For each of these n-uplets, a model will be calibrated with them and the validation loss value will be recorded. The selected n-uplet will be the one allowing to minimize this validation loss. This method is similar to the "brute force". Indeed, all combinations are tested without any particular reflection. However, it would be logical to think that the loss associated with two n-uplets which would be identical to one component, would not be very different especially if this hyperparameter is not very important. This method is therefore costly in time and calculation and is subject to what is called "the curse of dimensionality". Indeed, the number of calculations required increases exponentially with the addition of a hyperparameter. In fact if we consider n hyperparameters and for each of them 10 tested values, the number of calculations increases as 10^n. We can generalize by saying that for n hyperparameters subdivided into d discrete values, d^n calculations will be required.

4.1.2 Random search

This technique appeared in order to reduce the computation time required by the grid search as Bergstra and al. (2011) shown. The principle is simple: it consists in drawing at random the values of the hyperparameters which will constitute the various n-uplets which will be evaluated. This method has two drawbacks:

- High variance of the loss during the calculations due to the random choice of the hyperparameters.
- Since the hyperparameters are chosen completely at random, luck still has a role in the performance of this method. Note that this effect decreases as the number of iterations increases.

However, due to its random nature, this method increases the chances of having analyzed the action space in a short time compared to the grid search. This probability increases all the more if the hyperparameters differ in importance. Let us note moreover that if we assume the existence of a global minimum for the validation loss, the grid search will probably only be able to provide a more or less good approximation of the hyperparameters associated with this global minimum, depending directly on the discretization grid used. On the contrary, with random search, the choice of the domain is not really imposed except for the extremities of each dimension. In order to better understand, the figure below illustrates the difference in the searching strategy between grid and random search. To make things clear we can imagine that we consider in a neural network the h_1 = "learning rate" and h_2 = "Ridge penalty". For each of them in the grid search, we would like to try three different values, for instance $h_1 \in \{0,01, 0,001, 0,0001\}$,$h_2 \in \{0, 1, 0, 01, 0, 001\}$. This boils down to test nine couplets $(h_1, h_2) \in \mathcal{X}$ where $\mathcal{X} = \{(0.01, 0.1), (0.01, 0.01), (0.01, 0.001),$
$(0.001, 0.1), (0.001, 0.01), (0.001, 0.001), (0.0001, 0.1), (0.0001, 0.01), (0.0001, 0.001)\}$ for the grid search. For the random search, only the extremities $\{(0.0001, 0, 001), (0.001, 0.1)\}$ are needed and random values will be chosen.

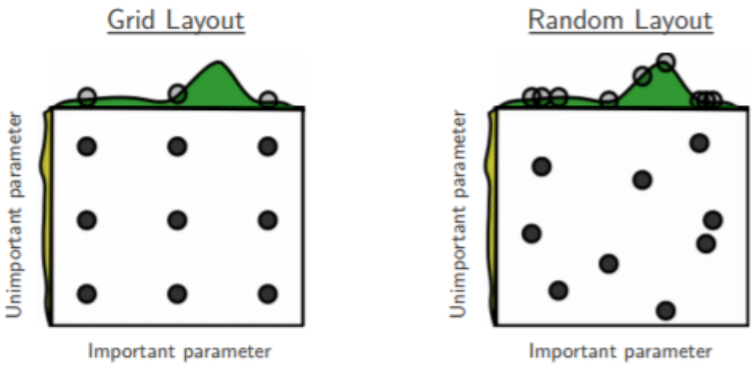


Figure 4.1.1: Grid and Random search comparison

4.1.3 Bayesian optimization

As the work Bergstra and al. (2011) have shown this optimization method is particularly useful when one has to evaluate an objective function $f : \mathcal{X} \to \mathbb{R}$ which is expensive to evaluate. It can therefore be worth to construct a less expensive function, which we will call the "surrogate" that tries to reproduce as well as possible the expensive function f. In our case we will choose $f = \mathcal{L}(x)$[1], our loss function [2]. This is in fact the idea of SMBO (Sequential Model Based Optimization) of which Bayesian optimization is a part. The goal is to spend more time intelligently looking for the next candidate x^*, the next hyperparameter vector, that we will test on the real costly objective function $f = \mathcal{L}(x)$ rather than trying a multitude of randomly chosen candidates[3] and spending a lot of time evaluating "bad" hyperparameters samples. Note that the surrogate is built on the basis of $\mathcal{H}$, containing the set of all the previous (x, y) pairs, i.e. the information already "seen" by the model. Note that here y refers to $f(x)$, so the loss associated to the hyperparameter vector x. You cand find below a general overview of how SMBOs work.

Algorithm 8 Sequential Model-based Global Optimization

1: $\mathcal{H} = \emptyset$
2: **for** t=0 to T **do**
3: $x^* = \underset{x \in \mathcal{X}}{\operatorname{argmin}}\ S(M_{t-1}, x)$
4: Evaluate $f(x^*)$
5: $\mathcal{H}_t = \mathcal{H}_{t-1} \cup (x^*, f(x^*))$
6: Fit a new model M_t to $\mathcal{H}_t$
7: **end for**
8: Return $\mathcal{H}$

Below, we will detail the step 3 of this algorithm, which consists in building the surrogate function. In order to determine x^*, we need a surrogate, a probabilistic model that we will call M to model our costly objective function $f = \mathcal{L}(x)$. This will allow us to calculate the Expected Improvement ($EI_{y^*}(x)$) which is defined as

$$\begin{aligned} EI_{y^*}(x) &:= \int_{-\infty}^{\infty} max(y^* - y, 0) p_M(y|x)\ dy \\ &= \int_{-\infty}^{y^*} (y^* - y) p_M(y|x)\ dy \end{aligned} \tag{4.1.2}$$

This corresponds to the conditional expectation, under a certain probabilistic model M, of f that $f(x)$ is smaller than a certain threshold y^*. There are different ways to calibrate M but we will limit ourselves here to describe the Tree-structured Parzen estimator (TPE) method that we will use in the section (6.1.4.2). Instead of modeling $p(y|x)$, we use Bayes' formula

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{4.1.3}$$

This one makes us go from $p(y|x)$ - "The probability of the score given the vector of hyperparameters"- to $p(x|y)$ - "The probability of choosing the vector of hyperparameters x knowing the score". Then, we define $p(x|y)$ with two densities

$$p(x|y) = \begin{cases} l(x) & \text{si } y < y^*, \\ g(x) & \text{si } y \geq y^*. \end{cases} \tag{4.1.4}$$

[1] Now, we consider our loss function as a function of the the hyperparameter vector x

[2] It is necessary here to consider the aggregated loss, i.e. $\mathcal{L}(x) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_x(y_i, \hat{y}_i)$

[3] We mean by "random" the fact of choosing them independently from the previous candidates x already tested, such as in the random search

or $l(x)$ is the density constructed from samples of hyperparameters x_i such that their corresponding loss $y = f(x_i)$ is less than some threshold y^*. Conversely, the density $g(x)$ is constructed on the basis of samples x_i that do not satisfy this constraint. Note that this algorithm depends on y^* but that it does not correspond to the lowest loss already observed $f(x^*)$. Indeed, y^* is chosen such that $y^* > f(x^*)$, so as to have enough points to build $l(x)$. The TPE algorithm will choose this threshold as being a certain quantile γ of the observed y values, so that $p(y < y^*) = \gamma$. Note that no specific model is needed for $p(y)$ has it will be shown in equation (4.1.6). With this framework, it is possible to optimize equation ((4.1.2)) in the TPE framework as follows

$$\begin{aligned} EI_{y^*}(x) &:= \int_{-\infty}^{y^*} (y^* - y) p_M(y|x)\ dy \\ &= \int_{-\infty}^{y^*} (y^* - y) \frac{p_M(x|y)p(y)}{p(x)}\ dy \end{aligned} \tag{4.1.5}$$

By construction, we know that $p(y < y^*) = \gamma$ and we can note that $p(x) = \int_{\mathbb{R}} p(x|y)p(y)dy = \gamma l(x) + (1-\gamma)g(x)$. This allows us to rewrite (4.1.5)

$$\begin{aligned} \int_{-\infty}^{y^*} (y^* - y) \frac{p_M(x|y)p(y)}{p(x)}\ dy &= \frac{l(x)}{p(x)} \int_{-\infty}^{y^*} (y^* - y)p(y)\ dy \\ &= \frac{l(x)y^*\gamma - l(x)\int_{-\infty}^{y^*} yp(y)\ dy}{\gamma l(x) + (1-\gamma)g(x)} \\ &= \frac{l(x)\big(y^*\gamma - \int_{-\infty}^{-y^*} yp(y)\ dy\big)}{l(x)\big(\gamma + (1-\gamma)\frac{g(x)}{l(x)}\big)} \\ &= \frac{y^*\gamma - \int_{-\infty}^{-y^*} yp(y)\ dy}{\gamma + (1-\gamma)\frac{g(x)}{l(x)}} \\ &\propto \left(\gamma + \frac{g(x)}{l(x)}(1-\gamma)\right)^{-1} \end{aligned} \tag{4.1.6}$$

The numerator of (4.1.6) is developed by linearity of the integrals and by the fact that $\int_{-\infty}^{y^*} p(y)dy = \gamma$. This indicates that it is preferable to propose a sample x with a high probability under $l(x)$ and a low probability under $g(x)$. Note that in the Bayesian framework, the input domain $\mathcal{X}$ will be represented by probability distributions unlike the two previous methods where the domain is represented by a discrete grid. These distributions will be our "a priori" and will be updated during the tests by densities estimated in a non-parametric way thanks to regression trees. These trees allow, by their construction, to treat both continuous and categorical variables, making them a candidate of choice. They are often coupled with bagging techniques to improve the stability of the results. Prior higher probability will be attributed to the x values whose probability of improvement of the y loss value is high. For example, the learning rate domain is generally considered prior lognormal.

Chapter 5

Efficient learning

5.1 Overfit and Underfit

In any regression problem it is important to distinguish the true signal from the noise. Indeed, when we try to calibrate a model, no matter the type, we try to explain the data in order to be able to predict future trends based on new data. Therefore, a model that reproduces the data, in other words that models the signal and the noise, is of little interest. This phenomenon is known as overfit.

There are different methods to avoid this kind of situation. One known method is to divide the data into different disjoint data sets, namely a training set, a validation set and a test set. The first one will be used to calibrate the model, it will be the data that the model will "see" directly and from which it will adapt its parameters in order to minimize the loss value as explained in the section (1.4). Note that this set often represents between 60% and 70% percent of the data. The second, the validation set, will allow to identify the moment when the model starts to overfit, in other words to model the noise. These data are not directly "seen" by the model during training, so it will not always adapt its weights in order to reduce the loss on this part of the data. Indeed, the loss on the validation set will initially be correlated to the loss on the training set and will therefore tend to decrease. However, the model will continue to adapt its weights in order to decrease the loss on the training set and will therefore inevitably start to model the noise and thus reproduce the data from the training set. At this point, the loss on the validation set will start to degrade, showing that the model begins to overfit. However, even if the validation set data are not directly seen by the network, it is important to be careful not to overfit them, but this will be detailed later in the practical part. In order to avoid this, the test set will allow to evaluate the model performance on new data which have not been used for the calibration of the weights nor for the hyper-parameters choice. This loss will give us an unbiased idea of the model performance. Then, this can be used to compare model between them.

Conversely, it is important not to underfit. This means that the model does not have enough degree of freedom, enough capacity of representation to allow it to generalize the data well. It is easier to remedy this situation by increasing the capacity of the model or by constraining it less. For example, in the case of a neural network, it is enough to increase the number of layers or the number of neurons per layer in order to give it more "memory" capacity. It is common to start with a model that overfits in order to be able to determine later the right compromise between generalization and data reproduction. Note that there are other methods that try to avoid overfitting, these will be discussed in the following sections.

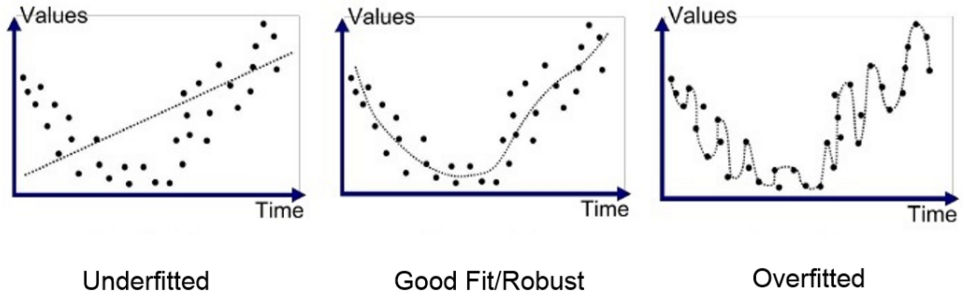


Figure 5.1.1: Learning goal

5.1.1 Regularization

In the following section we will present three of the most popular regularization techniques: Lasso ($L1$ regularization), Ridge ($L2$ regularization) and the dropout method. All these methods have the same goal, namely to determine the most parsimonious model while avoiding overfitting. However, these methods do not all have the same effect on the model parameters. These effects are directly linked to the nature of these methods. It is therefore essential to understand precisely how they act. Note that the first two methods can be applied to any model (neural networks, GLM, ...) while the last one, the dropout, is a method specific to neural networks.

5.1.1.1 Lasso: L_1 Regularization

This method adds a penalty to the loss function. This penalty is proportional to the $L1$ norm of the model's parameters. Let's assume a classical linear regression with p explanatory variables [1] the problem becomes

$$\beta^* = \underset{\beta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_\beta(\hat{y}_i, y_i) + \lambda \sum_{k=1}^{p} |\beta_k| \tag{5.1.1}$$

where $\beta \in \mathbb{R}^p$, $\lambda \in \mathbb{R}^+$ is the parameters vector and consider the case of a classical quadratic loss function so that $\mathcal{L}_\beta(\hat{y}_i, y_i) = (y_i - \hat{y}_i)^2$. In this case, without the penalty, the solution vector β^* is the one which satisfies the constraint $\frac{\partial \mathcal{L}}{\partial \beta} = 0$ and leads to $\beta^* = (X^T X)^{-1} X^T Y$. Unfortunately, the determinant of the matrix $(X^T X)$ is sometimes close to zero because of colinearity between explanatory variables or if the number of explanatory variables p is important compared to the sample size n. This affects the model performance due to overfitting. The adding of the L_1 penalty allows to keep the β_k with $k \in \{1, ..., p\}$ to be "small". To be more precise, the new loss function $\mathcal{L}_\beta^{L_1}(y_i, \hat{y}_i)$ corresponds to the Lagrangian of the initial problem under the constraint $||\beta||_1 \leq \gamma$ for some upper bound $\gamma \in \mathbb{R}^+$ on the L_1 norm of β. Note that the bigger λ is, the smaller γ is.

Moreover, an interesting effect of this constraint is to force some β_k to be zero. This means that the model is able to select by itself the most relevant variables. This phenomenon can be explained and visualized simply if we consider the 2D case as Denuit and al. (2019) have explained. Indeed, we can represent the problem as follows. On the one hand we have the constraint which can be represented in the plan as a square rotated so that its corners lie on the axes. On the other hand, the quadratic loss function can be represented as a family of ellipses $\mathcal{L}_\beta = d$ that would share the same center and the same ratio between their two axes[2]. Note that each point of a same ellipse represents all the (β_1, β_2) combinations that give the same loss value. Without the penalty, the minimum loss is attained at $\mathcal{L}(\beta^*)$, which is also the center of all the other ellipses of equation $\mathcal{L}(\beta) \geq d$.

[1] To simplify, consider that the intercept is in the p vector of explanatory variables, so $\mathrm{x}_{i,1} = 1 \ \ \forall i \in \{1, 2, ..., n\}$

[2] We consider here the two axes of symmetry of the ellipse.

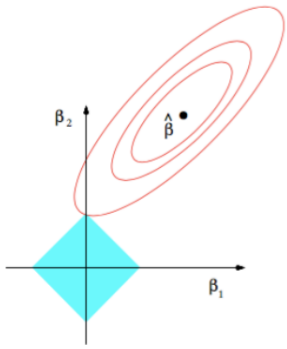


Figure 5.1.2: Lasso regularization

The β combination that minimizes $\mathcal{L}_\beta^{L_1}$ corresponds to the ellipse defined by the value of d such that this ellipse is tangent with the constraint boundaries. A convex object, like an ellipse, tangent to the constrained boundary is likely to encounter a corner of a hypercube for which some of the β values are identically equal to zero.

5.1.1.2 Ridge: L_2 Regularization

Like the $L1$ regularization, this method consists in adding a penalty term to the loss function with the difference that it is proportional to the $L2$ norm of the parameters vector β. If we consider a classical linear regression with p explanatory variables the problem becomes

$$\beta^* = \underset{\beta}{\operatorname{argmin}} \frac{1}{n}\sum_{i=1}^{n}(y_i - \beta^\intercal x_i)^2 + \lambda\sum_{k=1}^{p}\beta_k^2 \tag{5.1.2}$$

Again with $\lambda \in \mathbb{R}^+$. As in the previous case, the optimal vector β^* that minimizes the unconstrained loss is $\beta^* = (X^TX)^{-1}X^TY$. We are again confronted with the potential invertibility problem of $(X^TX)^{-1}$. The adding of the L_2 constraint changes the expression of β^* which now directly depends on λ. In fact, thanks to the first order condition $\frac{\partial \mathcal{L}_\beta^{L_2}}{\partial \beta} = 0$ we can write

$$\begin{aligned} &2X^\intercal(Y - X\beta) + 2\lambda\beta = 0 \\ &X'Y - X^\intercal X\beta + \lambda\beta = 0 \\ &\beta_{L_2}^* = (X^\intercal X + \lambda I)^{-1}X^\intercal Y \end{aligned} \tag{5.1.3}$$

where I is the $p \times p$ identity matrix and where $\lambda \in \mathbb{R}^+$. Now, the problem is well-posed because of the singularity and invertibility of $(X^TX + \lambda I)$. As before, the equation corresponds to the Lagrangian of the initial problem but this time under a constraint depending on $L2$ norm, namely $||\beta||_2 \leq \gamma$ for some upper bound $\gamma \in \mathbb{R}^+$ on the L_2 norm of β. Again, the bigger λ is, the smaller γ is. Note that adding a $p \times p$ identity matrix multiplied by λ is also called the Thikonov regularization. Note that we can also understand the effect of adding this L_2 penalty by computing the second derivative of the constraint $\frac{\partial^2(\lambda\beta^T\beta)}{\partial\beta^2} = 2\lambda$. Given that $\lambda \in \mathbb{R}^+$, this implies that choosing a large λ makes the problem more convex. Therefore, we can interpret the value of this parameter as the degree of correction to be applied to the initial problem so that it becomes well-posed. Note that the major difference between the L_1 and L_2 regularization is that the latter will not set some parameters to zero even if it allows to keep them "small" as Melkumova and al. (2017) have highlighted. Again, we can understand this phenomenon using the 2D representation of Denuit and al. (2019). This time, the constraint is represented by a circle centered in zero. With this configuration, the points where some $\beta_k = 0$ for $k \in \{1, ..., p\}$ are not distinguished from the others. Therefore, the tangent point between the optimal ellipse and the constraint is no more likely to be a point where some $\beta_k = 0$. The figure 5.1.3 illustrates this.

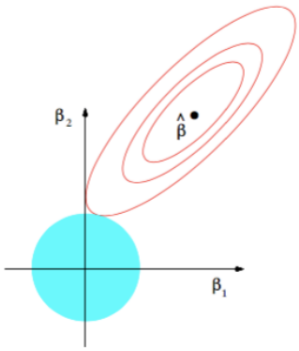
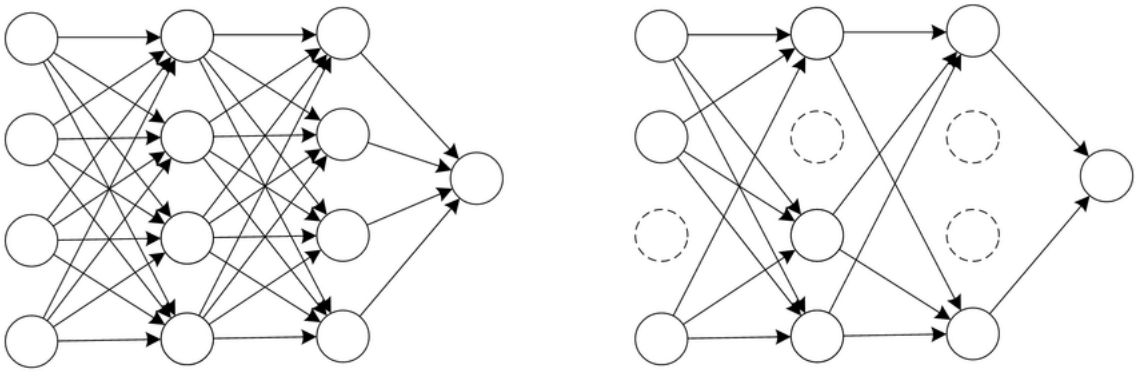


Figure 5.1.3: Ridge regularization

5.1.2 Dropout

The dropout technique was also developed by Hinton and al. (2014). It consists in deliberately turning off a certain proportion $0 \leq p \leq 1$ of the output of one or more given layers during training. By turn-off, we mean replacing them by zero. Note that p can vary from one layer to another, so we will note the dropout rate of layer l as $p^{[l]}$. As a result, the structure of the network changes as shown below.



(a) Standard Neural Network

(b) Network after Dropout

Figure 5.1.4: Dropout

The main idea behind this method is to avoid that some neurons take over the others. Indeed, we can see each layer of the network as a new input vector of dimension $(n^{[l]} \times 1)$ for the next layer $l+1$. Therefore, some components of this new vector can explain so well the output signal that they can take over the others during the training. Adding dropout will prevent the network from relying on only a few elements and so will prevent to overtrain them which actually causes overfitting. Thus, dropout will force all neurons to train and will help to limit overfitting. In idea, we observe an important similarity with random forests where a subset of the explanatory variables is randomly chosen for each node of each tree. However, it should be noted that when we evaluate the performance of the model on the validation data, no layer's output will be set to zero unlike during the training. Therefore, we will multiply by $p^{[l]}$ all the output of the layer l in order to take into account the fact that more neurons are active during the validation than during the training.

Chapter 6

Practical study

In this chapter we will use the elements presented in the previous sections to model temperatures using neural networks. First, we will model the daily average temperature of the city of Madrid (ie DAT). This is defined as the average between the highest and lowest temperature of the day. Unlike other processes, such as stock prices, temperatures are relatively stable, for example thanks to the presence of seasonality, allowing the use of data over longer periods of time. The major difficulty will be to get hold of high quality data. Indeed, unlike financial information, weather data is more difficult to access, especially in Europe. However, we should note the presence of ECAD (European Climate Assessment Dataset project[1]) on this market which provides a good amount of meteorological data (temperature, pressure, wind speed/direction, ...) for a large variety of cities in Europe. It is still necessary to ensure the integrity of the data collected, meaning to check the presence of absurd values or the presence of missing data. There are many methods to manage these specific cases as Zapranis and al. (2013) explained, but we did not use them because our data did not present this kind of problem. We will use the average temperature of the city of Madrid between 30/11/1986 and 30/11/2020 to calibrate univariate models and we will add the temperatures of Barcelona during the same period in order to calibrate a multivariate model. Note that the 29th of February of leap years have been removed to ensure a regular annual cycle of 365 days.

All these elements will allow us to simulate the distribution of different indexes on which we will price weather derivatives. Here are the different indexes that we will try to reproduce:

- CAT indexes: This is established as the sum of the daily average temperatures (DAT) over the contract period. As already mentioned, the DAT is calculated as the average of the highest and lowest temperature of the day. Note that this is what our models will predict and it is on this data that they will be trained. This index generally corresponds, in Europe, to the contract that extends over the summer months. In all the European countries, one CAT index futures contract pays off €20 per index point while it is £20 in London.

- HDD or CDD: The HDD index is established as the number of degrees of average daily temperatures (DAT) that are below a reference temperature. Conversely, the CDD is established as the number of degrees of average daily temperatures that are above a reference temperature. If we note this temperature threshold T^* and if we assume that the contract start at $t = 0$ and have a maturity T with $0 \leq t \leq T$, we can note

$$\begin{aligned} HDD_t &= \sum_{k=0}^{t} \max(0, T^* - DAT_k) \\ CDD_t &= \sum_{k=0}^{t} \max(0, DAT_k - T^*) \end{aligned} \tag{6.0.1}$$

[1] https://www.ecad.eu/

Generally $T^* = 65°F$ for the USA and $T^* = 18°C$ in Europe and Japan. Note that these contracts have generally a monthly or seasonal duration.

Derivatives on these indexes are usually vanilla options or future contracts. In the context of weather derivatives, a future will pay at maturity the value of the underlying index, regardless of its value, unlike options where the payoff will be truncated at the value of the strike K, predetermined at the time the contract is set up, $t = 0$. The payoff is therefore always positive or zero and will be worth

- Call $= \max(0, S_T - K)$
- Put $= \max(0, \ K - S_T)$

if we note S_T the value of the index considered at the maturity of the contract. Note that the payoffs are linear for $S_T \geq 0$ for the call and for $S_T \leq 0$ for the put. This means that these are a "one-sided" risk unlike futures contracts where the risk is bilateral.

6.1 Univariate Model

6.1.1 Model equation

As already explained in the introduction chapter, we will model the Madrid temperatures thanks to neural networks. First, it is important to be able to visualize the data in order to understand its different components.

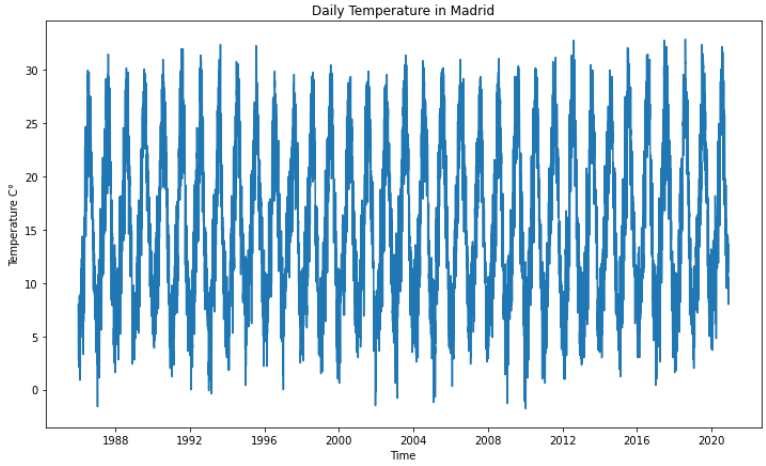


Figure 6.1.1: Daily average temperature of Madrid

As usual with temperature data, we directly observe a cycle where each spike corresponds to a year of data. This seasonality needs to be modeled and it seems natural to model this cycle with sines or cosines for instance. Moreover, there may be additional cycles in the data (monthly, weakly pattern,...) but not visible at first sight on this figure. This justifies the idea of potentially using not only one but several sines/cosines or even wavelets as Zapranis and al. (2009) did. However, the latter goes beyond this thesis.

In order to model this time series we will use the classical assumptions which are made for this type of data, such as the ones made by Zapranis and al. (2009). It usually assumes a predictable seasonality subject to global warming around which the temperatures oscillate with higher volatility during the winter than the summer. Early models were using AR(1) processes (cfr appendix (8.6)) or continuous equivalent such as Benth and al. (2005). Even if these models fail to capture the slow time decay of the autocorrelation of temperatures, we will build a model on the general idea of an AR(p) process (cfr appendix 8.6).

We will model the temperatures as follows

$$T_t = S_t + f_t(T_{t-1}, ..., T_{t-p}) + \epsilon(t) \qquad 0 \leq t \leq T \tag{6.1.1}$$

$$\tilde{T_t} = f_t(T_{t-1}, ..., T_{t-p}) + \epsilon(t) \qquad 0 \leq t \leq T \tag{6.1.2}$$

It is therefore assumed that the temperature at a certain time t [2], can be explained by the sum of two deterministic and one stochastic components, respectively

- S_t the trend and the seasonality at time t.
- $\mathbb{E}(\tilde{T_t}|\tilde{T}_{t-1}, ..., \tilde{T}_{t-p}) = \hat{f}(\tilde{T}_{t-1}, ..., \tilde{T}_{t-p})$ the conditional expectation of the detrend temperature $\tilde{T_t} := T_t - S_t$ at time t, which is a function of the detrend temperatures of the last p days with $p \in \mathbb{N}^*$.
- $\epsilon(t) = \sigma(t)u(t)$ where $\sigma(t)$ is the volatility function which is assumed seasonal with an annual cycle[3] and where we make the hypothesis that $u_t \sim \mathcal{N}(0,1)$ for $t \in [0,T]$.

This assumes that $T_t \sim \mathcal{N}\Big(S_t + f_t(T_{t-1}, ..., T_{t-p}), \sigma_t^2\Big)$ at time t. We must be careful not to confuse conditional and unconditional distributions. The distribution is here conditional on time $t \in \big[0,T\big]$ of the observation and we have emphasized that $\sigma(t)$ has a one year cycle. This means, for example, that the noise of the temperatures of 21/01 (whatever the year) will have an identical distribution, namely $\epsilon_t \sim \mathcal{N}(0, \sigma_t^2)$. Moreover, a normal conditional distribution does not necessarily lead to a normal unconditional distribution - unconditional in the sense of the time.

We also assume that the seasonality S_t and the volatility function $\sigma(t)$ are deterministic functions modeled as follows

$$S_t = \underbrace{\lambda + \beta t}_{\text{Trend}} + \underbrace{\sum_{i=1}^{N_1} a_i \ sin\Big(\frac{2i\pi(t-f_i)}{365}\Big) + \sum_{j=1}^{N_2} b_j \ cos\Big(\frac{2j\pi(t-g_j)}{365}\Big)}_{\text{seasonality}} \tag{6.1.3}$$

$$\sigma(t) = \eta + \sum_{i=1}^{J_1} c_i \ sin\Big(\frac{2i\pi(t-\rho_i)}{365}\Big) + \sum_{j=1}^{J_2} d_j \ cos\Big(\frac{2j\pi(t-\gamma_i)}{365}\Big) \tag{6.1.4}$$

We will therefore try to estimate $f_t(T_{t-1}, ..., T_{t-p})$ using different neural networks in order to build our estimation $\hat{f}_t(T_{t-1}, ..., T_{t-p}) \approx f_t(T_{t-1}, ..., T_{t-p})$. We will also estimate the function $\sigma(t)$ in order to be able to build a model from which we will make plausible simulations. This will be necessary in order to correctly price weather derivatives.

6.1.2 Trend and Seasonality calibration

As mentioned in the previous section, the trend and seasonality S_t is modeled thanks to a sum of sines and cosines with a yearly frequency and a linear trend, such as Zapranis and al. (2009). The latter can be interpreted by different effects such as global warming. The parameters will be estimated by minimizing the square differences between our assumed model $\hat{S}_t$ and the corresponding true temperature y_t. This leads to the following problem

$$\Phi^* = \underset{\Phi}{\text{argmin}} \ \frac{1}{n} \sum_{i=1}^{T} (y_i - \hat{S}_i(\Phi))^2 \tag{6.1.5}$$

[2] We consider that $[0,T]$ corresponds to the dataset time range.

[3] That imposes each year of data has the same number of days, justifying therefore the withdrawal of the 29/02 of the leap years

Where Φ is the parameters vector. Minimizing the squared residuals is the same as maximizing gaussian log-likelihood (cfr appendix 8.2). Therefore, we know that our estimators asymptotically follow a normal distribution because of the MLE properties. Thanks to this remark, we can build confidence interval. Note that we also have to find the right couple (N_1, N_2) that minimizes the best our loss function. We have therefore calibrated different models leading to the choice of using two sines to model our seasonality, so $(N_1 = 2, N_2 = 0)$. This can be seen with trials and errors but also with the discrete Fourier transform of the times series which presents two spikes at angular frequency corresponding to annual and biannual cycle. Find below in table (6.1.1) the parameters estimates with a 95% confidence interval

Parameters	$[\text{IC}_{95\%}$	Estimate	$\text{IC}_{95\%}]$	Standard deviation
λ	14.68	14.78	14.88	5.28×10^{-2}
β	8.23×10^{-5}	9.63×10^{-5}	1.10×10^{-4}	7.18×10^{-6}
a_1	-9.74	-9.67	-9.60	3.74×10^{-2}
f_1	657.44	657.88	658.32	2.24×10^{-1}
a_2	1.6	1.67	1.74	3.73×10^{-2}
f_2	-5.45	-4.17	-2.9	6.50×10^{-1}

Table 6.1.1: Trend and seasonal parameters

We can see that none of the above confidence intervals contains zero. This means that they are all significant according to a t-test at a level $\alpha = 0.05$ for the hypothesis test $H_0 : \beta = 0$, if we note the parameter tested β. In other words, $p_{value_\alpha} < 0.05$ for each of the above parameters. The following figure (6.1.2) illustrate the results of the calibration of $\hat{S}_t \approx S_t$ from equation (6.1.5)

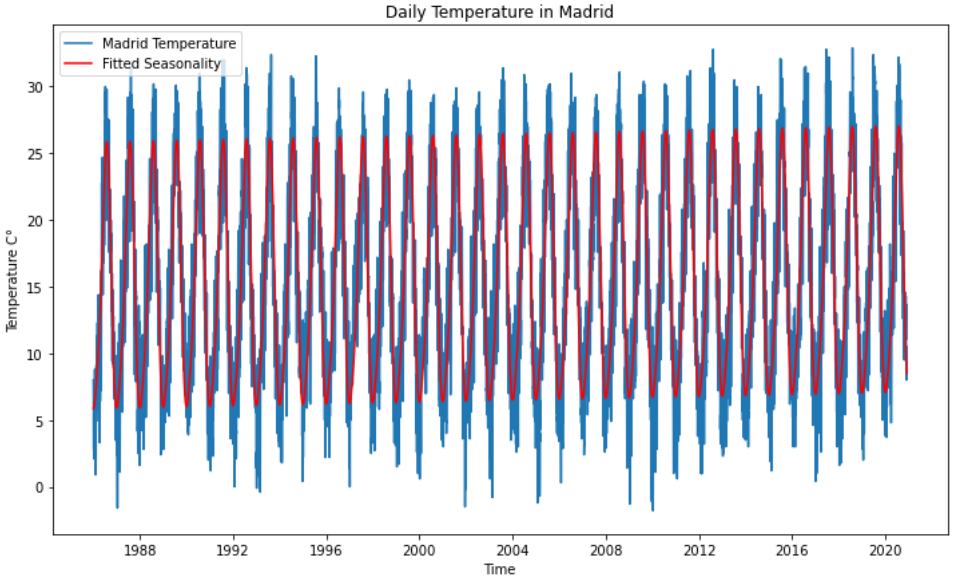


Figure 6.1.2: Fitted trend and seasonality

Visually, the seasonality seems to fit the data well. In order to observe the benefit of adding the second sine, you will find below on figure (6.1.3) representing the average monthly residuals for $N_1 = 1$ and $N_1 = 2$. We can observe several interesting points on this figure. Firstly we notice that the original data presents a strong one year cycle that confirms to use at least on sine. Secondly, when the first sine has been applied, a biannual cycle appears in the monthly residuals, thus confirming the decision to choose $N_1 = 2$ rather than $N_1 = 1$. Moreover, when adding the second sine, we observe a crushing of the curve translated by a decrease of the standard deviation from 1.43 to 0.70 degree for $N_1 = 1$

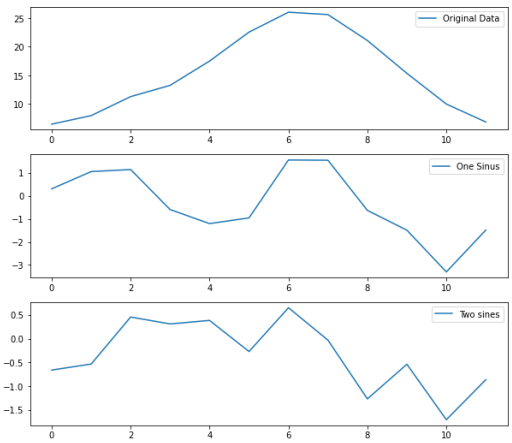


Figure 6.1.3: Monthly mean of $\tilde{T}$

and $N_1 = 2$ respectively while the mean is closer to 0. We could add additional frequencies in order to test potential tri-annual, weekly cycles,... but adding an extra sine, corresponding to a potential quarterly cycle, degrades the loss value leading us to stop at $N_1 = 2$. We can also notice on figure (6.1.3) than the linear upward trend also fits well the data. Note that it would have been possible not to define a parametric form as done in equation (6.1.3). We could have added explanatory variables to the models by indicating for each temperature: the year, the month, the week, the quarter,... of this observation. Then, the network could have modeled the trend and the seasonality based on this information and maybe even detect some cycles that would not be captured by (6.1.3). However, we did not explore this possibility for two main reasons:

- First, this would strongly increase the number of explanatory variables and therefore the sizes of the neural networks as well as the calibration time. Indeed, as explained in the appendix (8.1), it would already take 364 additional binary variables to indicate to which day an observation belongs.
- The model would lose interpretability. The benefit of the parametric form presented in equation (6.1.3) is that this component can be isolated from the rest.

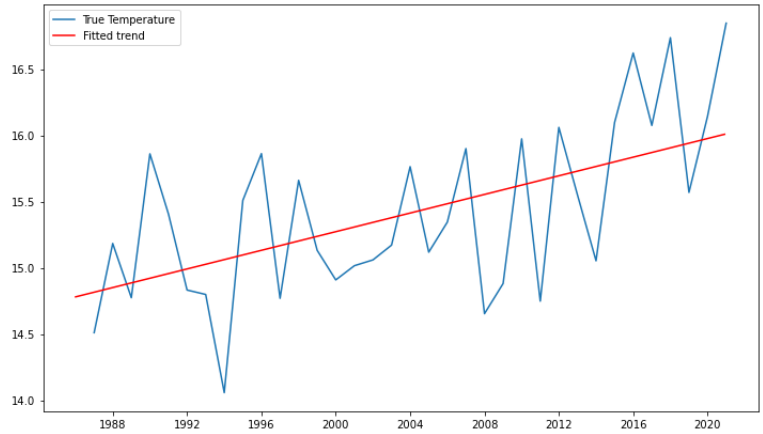


Figure 6.1.4: Yearly mean temperatures

We therefore recover $\tilde{T}_t = T_t - S_t$ the detrend and deseasonalized temperatures. This is the data of interest that we will try to model thanks to neural networks. We make the assumption that this variable is conditionally normally distributed with mean zero and with variance that depends on the day in the year no matter the year. Moreover, we assume this volatility function repeats itself every year.

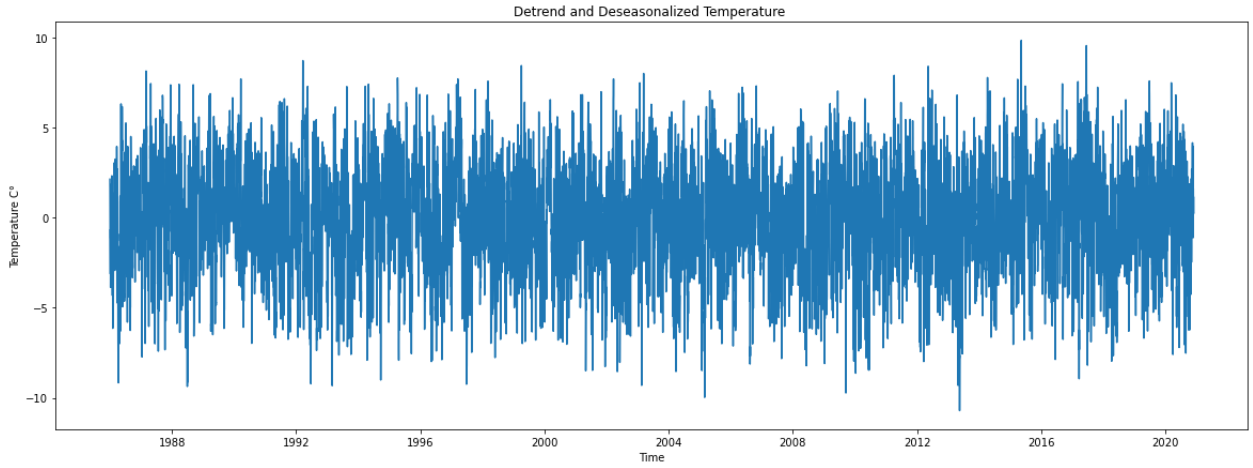


Figure 6.1.5: Detrend and deseasonalized temperatures $\hat{T}$

We can also check for the stationarity of this time series thanks to the Dickey-Fuller test (cfr appendix 8.5). The time series is well stationary ($p_{value_{\alpha=0.05}} < 0.05$) for different subdivisions of the data allowing to hope to calibrate a good predictive model.

6.1.3 Benchmark model for Detrend and deseasonalized

In order to be able to evaluate the gains brought by the use of neural networks in our temperature model, it is wise to use a reference model, preferably simple. Thus, we can evaluate the interest of more complex models in view of the improvement that they bring. This "benchmark" model will be an AR(p) (cfr appendix 8.6). It is a discrete model meaning that $(\tilde{T}_t)_{t\in\mathbb{N}}$. It has the following form:

$$\tilde{T}_t = \omega + \sum_{i=1}^{p} \rho_i \tilde{T}_{t-i} + \epsilon_t \qquad 0 \leq t \leq T \tag{6.1.6}$$

where ρ_i with $i \in \{1, ..., p\}$ are estimated by log-likelihood maximisation and where p is fixed by trial and error. Note that the $\epsilon_t \sim \mathcal{N}\left(0, \sigma^2\right)$ is the noise term. As explained in section (5.1) we decide to divide our data into three disjoint sets. The training set will contain the first 60% of the data while the validation and the test sets will contains respectively the next 20% and the next 20% of the data. This setting will also be used for all the following models that will be built. We present below in table (6.1.2) the results of calibration for our best model

Parameter	[IC$_{95\%}$	Estimate	IC$_{95\%}$]	Standard deviation
ρ_1	0.929	0.9507	0.973	0.011
ρ_2	−0.255	−0.2254	−0.195	0.015
ρ_3	0.029	0.0505	0.072	0.011

Table 6.1.2: AR(3) parameters

Which leads to our AR(3) model equation

$$\tilde{T}_t = 0.9507\tilde{T}_{t-1} + -0.2254\tilde{T}_{t-2} + 0.0505\tilde{T}_{t-3} + \epsilon_t \qquad 0 \leq t \leq T \tag{6.1.7}$$

We also tried higher value for p but it resulted in insignificant[4] parameters leading to choose the AR(3) as a good benchmark. We also observed that ω was never significant. This makes sense because the data oscillates around zero, showing that the previous section operations work well. Note that the residuals of this models show no more serial correlation because we do not reject H_0 : - "There is no autocorrelation in the tested series" - of the Box-Ljung test (Box-Ljung statistic = 5.79 , $p-value_{\alpha=0.95} > 0.05$). More information about this test has available at appendix (8.4). However, the residuals do not respect the hypothesis that $\epsilon_t \sim \mathcal{N}\left(0, \sigma^2\right)$ as indicated by the Jarque-Bera statistics ($JB_{stat} = 383.842, p_{value_{\alpha=0.05}} < 0.05$) (cfr appendix 8.3). It seems to confirm the idea of assuming $\epsilon_t = \sigma_t u_t$ as suggested in equation((6.1.1)) and thus to assume a conditional normality instead of an unconditional normality of $\tilde{T}_t$. Table (6.1.3) shows the performance of our AR(3) model on the test set.

p	p=1	p=2	p=3
Test set loss	0.3867	0.3739	0.3735

Table 6.1.3: Test set loss

This informs us that a performance of at least 0.3735 on the test set is expected from our neural networks in order to be chosen to model $\tilde{T}$.

6.1.4 Neural Networks univariate modeling

6.1.4.1 Preprocessing

Before being able to calibrate our neural networks, it is important to perform some transformations on our variables. This step is called "preprocessing" and it is essential. The preprocessing concerns a wide range of methods allowing for example to reduce the dimensions of the space of the explanatory variables, to transform the categorical variables into something interpretable by the model (cfr appendix 8.1) or to change the range of the different explanatory variables,... It is this last application that interests us here. In fact, this benefits the convergence speed of the model by allowing to keep all its parameters (i.e. the weights and biases) in the same range of values as Virili and al.(2007) explained. This is therefore essential when faced with a problem with explanatory variables whose values differ greatly. In fact, given that we use the error between predictions and targets and the use of small weights, the scale of input and output is an essential factor. It is better that the model learns small weights in order to be stabler when predicting with it. In fact, unscaled input variables can induce slow or unstable learning process or even an exploding gradient causing the fail of the learning process.

Consider that we have n vector $x_i \in \mathbb{R}^p$ of p explanatory variables, then we can standardize the data as follows

$$x^*_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j} \qquad \forall i \in \{1, ..., n\},\ \forall j \in \{1, ..., p\} \tag{6.1.8}$$

where μ_j is the mean of x_j and σ_j its standard deviation. This operation can also be made for the target variable $y_i \in \mathbb{R}$. This method assumes that we can estimate well these quantities thus implying the normality of the data. This allows to restrict the possible interval for the variable but it does not guarantee the range $[-1, 1]$. The obtained range will be conditional on the respect of the underlying normality assumption, especially the one of the tails. Note that using the classical formulas $\hat{\mu_j} = \frac{1}{n}\sum_{i=1}^{n} x_{i,j}$ and $\hat{\sigma_j} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_{i,j} - \mu_j)^2}$ is to use the formulas that maximize the log-likelihood of a normal distribution. In fact, these are MLE estimators of these quantities. Then it can be wise to use others techniques when the normality does not apply. There are a lot of other methods but the

[4]Note that when we refer to (in)significant coefficient, this is in the sens of the t-test.

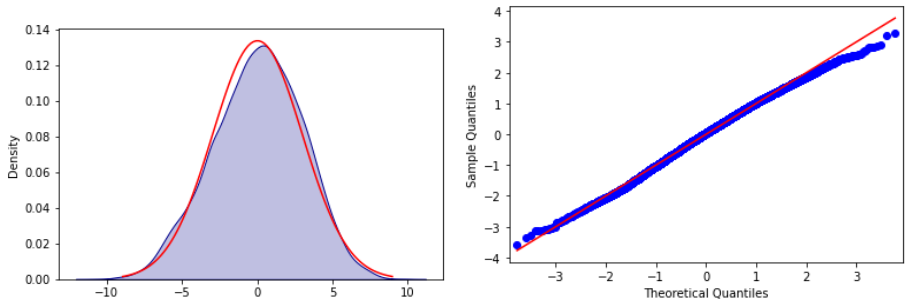
most second frequent one is the min-max normalisation. This corresponds to

$$x^*_{i,j} = \frac{x_{i,j} - min_j}{max_j - min_j} \ \forall i \in \{1, ..., n\}, \ \forall j \in \{1, ..., p\} \tag{6.1.9}$$

This method guarantees that $x^*_j \in [0, 1] \quad \forall j \in \{1, ..., p\}$ if the minimum and maximum are well estimated. In fact, if the model is used in the future to make predictions and that one of the new data exceeds the maximum on which the model has been trained, the risk is to make unreliable predictions.

Chollet and al. (2018) draw attention to the fact that, no matter the method, we have to compute these values[5] on the training set only. This allows not to communicate any information of validation or test data during the training.

We decided to choose the first method given that the empirical distribution of $\tilde{T}_t$ is not so far from a $\mathcal{N}(0, 8.91)$. Note that after the transformation explained in (6.1.8), $\tilde{T}_t$ follows approximatively $\mathcal{N}(0, 1)$. However, this allows us to work with a well known distribution while limiting the effect of this approximation. This is regularly done in other research on temperature modeling such as in Zapranis and al.(2009). In order to be more accurate, it would be possible to use a deviance of a distribution that fit better the data. But this would bring complexity for probably little improvement since the empirical distribution deviates only a little from a Normal distribution.



(a) density $\tilde{T}$ (b) QQ plot of $\tilde{T}$

Figure 6.1.6: Standardized residuals normality

This leads us to choose a mean square error as loss function. Indeed, as explained in the appendix (8.2), it corresponds to the Gaussian deviance. The minimization of this metric is therefore equivalent to maximizing the Gaussian log-likelihood. Moreover, it is important to be able to compare the loss of the benchmark model with those of the models we will calibrate.

6.1.4.2 Neural Networks building

After normalizing the data, we can start building our neural networks. We will start with fully connected neural networks (no recurrent networks for the moment), to model $\left(\tilde{T}_t\right)_{t\in[0,T]}$[6]. The purpose will be to model the temperature of the next day, so $\tilde{T}_{t+1}$ according to the temperatures of the last p days $\{\tilde{T}_{t-1}, ..., \tilde{T}_{t-p}\}$. This means that we have to fix the number of lags p to take into account. As explained in section (1.3.2), this implies the representation of each training example in a single point. The potential importance of the order of arrival of $\{\tilde{T}_{t-1}, ..., \tilde{T}_{t-p}\}$ will not be captured by this kind of network.

[5] We refer to μ_j, σ_j, min_j, max_j

[6] For the ease of notation, when building the neural networks, $\tilde{T}$ corresponds to the scaled detrend and deseasonalized temperatures

We will therefore test several lags in order to determine the optimal sequence length p. A first approach could be to use the classical tools for modeling time series with Arma models such as autocorrelograms or partial correlations. However, correlation is a measure of linear dependence, so it cannot capture potential non-linear dependence effects, unlike neural networks. Indeed, the latter, thanks to the activation functions, can introduce non-linear dependency effects. This implies that these tools are merely indicative but not determinant in deciding the number of lags to take into account. Moreover, we expect that $p \geq 3$ given the benchmark model results.

The strategy for building neural networks will be the following:

- First, we will determine the optimal number of hidden layers. This number logically depends on the number of neurons assigned to each layer. We will therefore fix these on a case by case basis following a first phase of trial and error. To do this, we will look at the number of layers that will minimize the validation error [7]. This one will be determined with the help of bag fraction of 80% in order to have an idea of the sensitivity of the performance to the training data. This is in the idea of bagging methods allowing to accelerate the calculation time and the stability. We prefer this method to the cross validation one in order to avoid temporal information leakage between the different data sets as explained Chollet (2019).

- In order to be as robust as possible against overfitting, it is important to ensure that the weights keep reasonable sizes and that their distribution has low entropy. As explained in section (5.1.1), adding a penalty allows a more regular distribution of weights. In this idea, we will try to add regularization in a second step. We will therefore test different types of regularization presented in section (5.1.1). We will choose the one that allows us to minimize the validation error. This step will be made with the help of Bayesian optimization (cfr section 4.1.3) to estimate the most promising penalty domain in which to search for.

- In a last step, we will divide the interval determined in the previous step into 9 equidistant values to fix our final penalty with a grid search. This optimal penalty is the one that will minimize once again the validation error. In order to ensure the stability of the results and to limit the effect of randomness present in these models during calibration, we again introduce bag fraction (again 80%) here. This will allow us to find the most parsimonious model possible in the hypothesis domain established by the tested architecture.

- Finally, in order to verify that we are not overfitting the validation set, we will calculate the loss value of the test set hoping to have a similar performance to the validation set, thus indicating the performance of the model on unseen data. In fact, given that all the hyperparameter choices are made given the validation error, we have to take care not to overfit it. This is why the use of a test set is a best practice.

Let us note moreover that the training examples will be shuffled before being processed by the neural network. Indeed, given that the consecutive examples are identical to one component, mixing them will allow a faster convergence of the model. In fact if we compare two consecutive learning examples, they will have the following form $\{(\tilde{T}_{t-2}, \tilde{T}_{t-1}, \tilde{T}_t, \tilde{T}_{t+1}), (\tilde{T}_{t-1}, \tilde{T}_t, \tilde{T}_{t+1}, \tilde{T}_{t+2})\}$.

We will finally choose the model with the smallest loss value on the test set. Note that we will not use the AIC and BIC criteria in the choice of the optimal model given the large number of parameters in neural networks. In order not to be redundant, we will have for each of the steps presented above all the calibration results of the neural networks for $p \in \{1, 3, 5\}$. We will comment on these from a global point of view as well as from an individual point of view. By "global" we mean the common points that can be put forward between the calibration of the models for different values of p. We will also introduce the notation "NN(.)" to represent a feed forward neural network (dense fully connected) where each of the components except the last one will refer to the number of neurons on the hidden

[7] As for the benchmark: training set: 60%, validation set: 20% and test set 20%

layer in question. The last component represents the number of neurons on the output layer and will therefore always be equal to 1 in our case. To be clear, a NN(3,1) has only one hidden layer of 3 neurons while NN(3,3,1) has a second hidden layer of 3 neurons. Apart from the number of layers and neurons per layer, all these different models have common characteristics, namely

- The activation functions used are each time the Relu activation (cfr section 1.2.3) except on the last layer which comprises a linear activation in order not to restrict the domain of the estimation. This choice is not insignificant because the Relu activation function allows to solve several problems encountered by other functions such as tanh and sigmoid during training. Indeed, these two functions suffer from vanishing gradient due to the expression of their derivative. This implies that when a neuron is strongly or very weakly activated, the gradient is very close to 0, thus slowing down or even preventing learning. Moreover, the Relu function is very simple and does not require any complex transformation unlike tanh and sigmoid and as often in machine learning, the simplest models are always considered as better, it is the principle of parsimony.

- The training data are given by batch of 256 to the neural networks and this during 700 epochs. This means that the weights are updated 31 times per epoch. The value 256 was chosen by trial and error because it allowed a relatively stable training without too much noise.

- We will also use the Adam algorithm to train these different models since it is the most complete of the optimizers presented in section (2.3). For the sake of awareness, we have first tried its faithful competitor RMSprop which provided similar results but slightly noisier. So we continued with Adam. As already explained, we also decided to fix the bag fraction at 80 % to introduce some variance in the training data. However, for the sake of comparison, each fold is the same for each of the different architecture. In other words, "fold 1" is always the same for NN(3,1),...,NN(3,3,3,3,1) when $p = 1$.

You can find below the training summary for all the tested models for $p = \{1, 3, 5\}$ respectively.

Lag 1								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(3,1)	0.3652	0.3738	0.3740	0.3738	0.3733	0.3741	0.0003	10
NN(3,3,1)	0.3646	0.3735	0.3742	0.3736	0.3725	0.3738	0.0006	22
NN(3,3,3,1)	0.3644	0.3743	0.3768	0.3732	0.3721	0.3751	0.0018	34
NN(3,3,3,3,1)	0.3643	0.3747	0.3752	0.3748	0.3731	0.3760	0.0010	46

Table 6.1.4: Training summary Lag 1 Models

Lag 3								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(5,1)	0.3501	0.3619	0.3625	0.3627	0.3612	0.3611	0.0008	16
NN(5,5,1)	0.3493	0.3610	0.3620	0.3605	0.3605	0.3611	0.0006	46
NN(5,5,5,1)	0.3470	0.3628	0.3650	0.3630	0.3602	0.3630	0.0017	76
NN(5,5,5,5,1)	0.3443	0.3644	0.3618	0.3614	0.3617	0.3625	0.0014	106

Table 6.1.5: Training summary Lag 3 Models

The selected models for each value of p are the models NN(3,3,1),NN(5,5,1) and NN(6,6,1). We can observe that whatever the value of p

Lag 5								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(6,1)	0.3469	0.3638	0.3660	0.3635	0.3613	0.3646	0.0017	16
NN(6,6,1)	0.3440	0.3632	0.3658	0.3620	0.3609	0.3643	0.0019	63
NN(6,6,6,1)	0.3404	0.3637	0.3681	0.3606	0.3627	0.3634	0.0027	161
NN(6,6,6,6,1)	0.3374	0.3642	0.3659	0.3625	0.3640	0.3645	0.0012	271

Table 6.1.6: Training summary Lag 5 Models

- the most adapted architecture is the one with two hidden layers. Indeed, it is this architecture that allows to reach the minimal validation error.
- Once the optimal structure is exceeded, the dispersion of the validation loss value increases, indicating rapid overfitting of the model. Indeed, if the capacity (i.e. the number of weights and thus the learning capacity of the model) is too large, one epoch will probably already be enough to overfit the training set. Moreover, as one epoch represents approximately 31 changes in the weights and the value of the validation loss is only given at the end of each epoch, the variance of the validation loss increases.
- We observe that when p increases, the performance on the training set get betters. However this improvement is not due to the addition of a new lag since the validation error does not improve. We can therefore attribute this performance gain to the increase of the network capacity leading to overfitting.

If we now consider each value of p individually, we note that

- $p = 1$: We can observe that the training loss decreases and seems to converge as we add layers while the validation loss reaches its lowest value for the NN(3,3,1) model. We also observe that even with additional neurons on each layers, the performance on the training set does not improve. This attempts to show that the residuals loss is the incomprehensible loss that cannot be overcome except by adding an explanatory variable or even more capacity. This can explain the slowdown of the training loss in table (6.1.4)
- $p = 3$: It is therefore this value of p that gives the best validation results, in particular the model with two hidden layers. Moreover, this model presents the lowest standard deviation of the four tested architectures. Moreover, since our validation loss is lower than the one of the benchmark, we can deduce that the link that exists is not linear. Therefore, a more constrained model such as the benchmark could only capture part of the relevant information resulting in a higher validation loss.
- $p = 5$: We observe that whatever the structure considered, the validation error does not decrease, thus indicating that the relevant information needed to predict the output is concentrated in the first 3 lags. This is in line with what the benchmark model showed. Then, we really fast overfit the data because we add capacity, one more neuron per layer, but we do not increase the relevant information, given that the three first lags contain the needed signal.

We now present the results of the Bayesian search for penalty estimation $\lambda_{L_1}, \lambda_{L_2}$ and the subsequent grid search. Note that the validation loss reported here is not computed on multiple models calibration as in the above analysis. This implies to take care of this metric because of its potential volatility. The real purpose here is just to determine the optimal order of magnitude of $\lambda_{L_1}, \lambda_{L_2}$.
We can see that for the three architectures $\lambda_{L_1} < \lambda_{L_2}$. This is due to the scale of the weights parameters. Given that they are for the majority, less than 1 in absolute value, their squares are less than their absolute values. This can explain this inequality. We also find that $L1$ regularization provides better validation performance in all three cases. Based on these results, we did a grid search around these points to test others penalties. Note that here we calibrated four folds for each value of λ_{L1} in order to have a more accurate idea of the models performance.

Models	Lasso	λ_{L_1}	Ridge	λ_{L_2}
NN(3,3,1)$_{p=1}$	0.3740	1.85×10^{-4}	0.3742	1.47×10^{-2}
NN(5,5,1)$_{p=2}$	0.3612	1.15×10^{-4}	0.3615	1.05×10^{-2}
NN(6,6,1)$_{p=3}$	0.3635	1.14×10^{-4}	0.3678	1.21×10^{-2}

Table 6.1.7: Bayesian search results for $\lambda_{L_1}, \lambda_{L_2}$

Lag 1 Regularization									
λ_{L1}	1×10^{-4}	$1{,}5\times10^{-4}$	2×10^{-4}	$2{,}5\times10^{-4}$	3×10^{-4}	$3{,}5\times10^{-4}$	4×10^{-4}	$4{,}5\times10^{-4}$	5×10^{-4}
Validation perf	0.3743	0.3746	0.3764	0.3759	0.3757	3759	0.3662	0.3664	0.3773
Standard deviation	0.0009	0.0013	0.0013	0.0010	0.0013	0.0008	0.0011	0.0011	0.0014
Fold 1	0.3755	0.3761	0.3679	0.3776	0.3767	0.3758	0.3770	0.3773	0.3785
Fold 2	0.3740	0.3741	0.3661	0.3756	0.3750	0.3763	0.3770	0.3771	0.3780
Fold 3	0.3731	0.3727	0.3645	0.3749	0.3591	0.3746	0.3743	0.3746	0.3749
Fold 4	0.3750	0.3754	0.3672	0.3755	0.3616	0.3769	0.3763	0.3764	0.3777

Table 6.1.8: Grid search results λ_{L_1} for NN(3,3,1)

Lag 3 Regularization									
λ_{L1}	1×10^{-4}	$1{,}5\times10^{-4}$	2×10^{-4}	$2{,}5\times10^{-4}$	3×10^{-4}	$3{,}5\times10^{-4}$	4×10^{-4}	$4{,}5\times10^{-4}$	5×10^{-4}
Validation perf	0.3630	0.3637	0.3622	0.3636	0.3624	3634	0.3624	0.3637	0.3633
Standard deviation	0.0008	0.0007	0.0006	0.0006	0.0013	0.0006	0.0006	0.0007	0.0009
Fold 1	0.3655	0.3655	0.3642	0.3646	0.3662	0.3639	0.3630	0.3654	0.3646
Fold 2	0.3625	0.3638	0.3620	0.3643	0.3627	0.3646	0.3627	0.3624	0.3649
Fold 3	0.3611	0.3616	0.3605	0.3615	0.3591	0.3615	0.3618	0.3625	0.3616
Fold 4	0.3628	0.3742	0.3621	0.3641	0.3616	0.3639	0.3623	0.3646	0.3621

Table 6.1.9: Grid search results λ_{L_1} for NN(5,5,1)

Lag 5 Regularization									
λ_{L1}	1×10^{-4}	$1{,}5\times10^{-4}$	2×10^{-4}	$2{,}5\times10^{-4}$	3×10^{-4}	$3{,}5\times10^{-4}$	4×10^{-4}	$4{,}5\times10^{-4}$	5×10^{-4}
Validation perf	0.3630	0.3646	0.3643	0.3655	0.3643	3718	0.3649	0.3646	0.3653
Standard deviation	0.0036	0.0019	0.0010	0.0022	0.0006	0.0038	0.0011	0.0025	0.0015
Fold 1	0.3687	0.3674	0.3653	0.3665	0.3648	0.3748	0.3648	0.3670	0.3677
Fold 2	0.3590	0.3622	0.3627	0.3631	0.3640	0.3740	0.3646	0.3624	0.3646
Fold 3	0.3608	0.3638	0.3646	0.3639	0.3633	0.3659	0.3634	0.3618	0.3638
Fold 4	0.3632	0.3651	0.3621	0.3686	0.3749	0.3751	0.3667	0.36472	0.3653

Table 6.1.10: Grid search results λ_{L_1} for NN(6,6,1)

We conclude that adding regularization does not improve the performance of any of the best models. We therefore decide not to add it since the neural network construction strategy seems to have avoided overfitting. Here are the performances of each of the best models for $p \in \{1, 3, 5\}$ on the test set

Models	Validation set	Test set
NN(3,3,1)$_{p=1}$	0.3735	0.3740
NN(5,5,1)$_{p=2}$	0.3610	0.3613
NN(6,6,1)$_{p=3}$	0.3632	0.3638

Table 6.1.11: Test loss

The losses on the test set are similar to those on the validation set which is a sign that we did not overfit the data. The model with $p = 3$ delivers the best performance on the test set and will be used

for the rest of the analysis. Moreover, it beats the performance of the benchmark model which was 0.3735. We can now look at the predictions of our best model on the test set on figure (6.1.7).

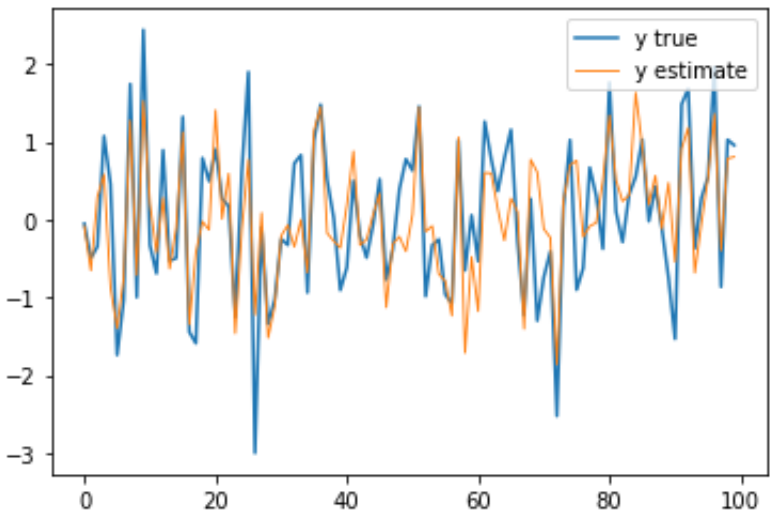


Figure 6.1.7: Predictions on the test set

We can also compare the predictions given by the best model for each value of p on figure (**??**).

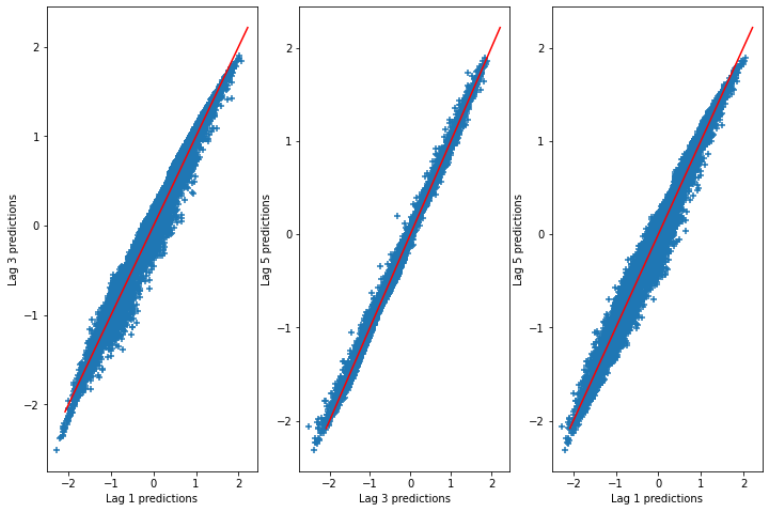


Figure 6.1.8: Scatterplot of predictions

We observe that the scatterplot opposing the predictions with the best Lag 3 and the best Lag 5 model is the one with the lowest variance around the line. This means that choosing p=5 in place of $p = 3$ does not change a lot the predicted values. This reinforces our choice concerning the most relevant value for p. We also can observe that when $p = 1$, the predictions tend to be lower in absolute value than the predictions with $p = 3$.

Once all the hyperparameters are fixed, we have to recalibrate our model on the full dataset in order to use all the relevant information for future predictions. It is also important to perform the reverse scaling operation (cfr equation 6.1.8) in order to recover our final predictions.

6.1.5 Residuals variance estimation

Recall that we made the assumption that our volatility function is of the following form

$$\sigma(t) = \lambda + \sum_{i=1}^{N} \phi_i \sin(\frac{2\pi\gamma_i t}{365} - \rho_i) \tag{6.1.10}$$

In order to be able to estimate our periodic function σ_t, we must first recover the residuals ϵ_t produced by our three-lags model NN(5,5,1), the one selected in the last section. Then, thanks to the variance formula $V(\epsilon_t) = E(\epsilon_t^2) - E(\epsilon_t)^2$, we can estimate $\sigma(t)$ with $\hat{\sigma}(t)$ as follows

- Split the observation in 365 groups, each corresponding to a single day in a year
- For each of these groups, compute $V(\epsilon_t) = E(\epsilon_t^2) - E(\epsilon_t)^2 \approx E(\epsilon_t^2)$ because $E(\epsilon_t)$ estimated by $\overline{X} = \frac{1}{n}\sum_{t=1}^{n} \epsilon_t = 0$
- Estimate $\sigma(t) = \sqrt{V(\epsilon_t)} \approx \sqrt{\frac{1}{n}\sum_{i=1}^{n} \epsilon_t^2}$ where n stands for the number of observation years, 35 in our case.
- Find by least square minimization the parameters

$$\underset{\lambda,\gamma,\phi,\rho}{\text{argmin}} \frac{1}{n} \sum_{i=1}^{365} (\hat{\sigma}(t) - \sigma(t))^2 \quad 0 \leq t \leq 365 \tag{6.1.11}$$

  where $\phi, \rho \in \mathbb{R}^N$.

We found that the value of N that minimizes the loss is equal to 4. This implies that $\gamma = \{1, 2, 3, 4\}$ and that $\phi, \rho \in \mathbb{R}^4$. The parameters values are available in table (6.1.12).

λ	ϕ_1	γ_1	ϕ_2	γ_2	ϕ_3	γ_3	ϕ_4	γ_4
1.750	0.032	0.168	5.658	-0.036	0.031	0.997	0.629	0.488

Table 6.1.12: Parameters for volatility seasonality

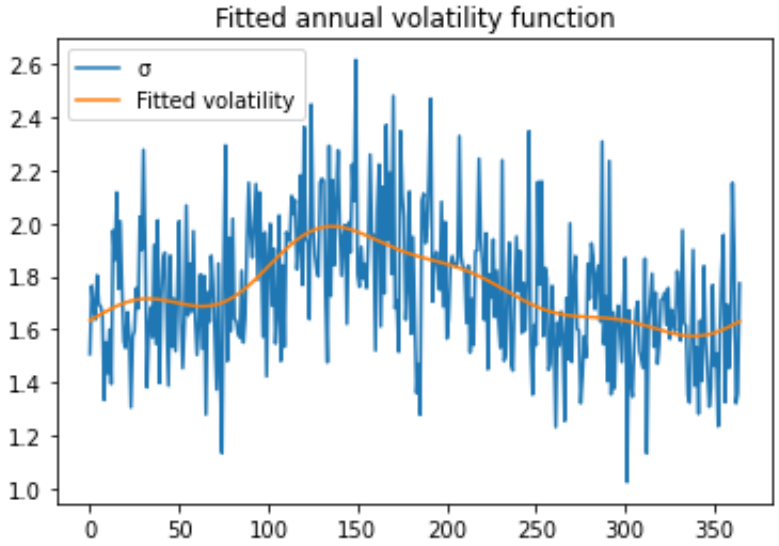


Figure 6.1.9: Fitted volatility seasonality

If we consider the volatility function as described in (6.1.10) and isolate $u(t)$ by standardizing our residuals with the $\hat{\sigma(t)}$ function, we should observe that $u_t = \frac{\tilde{T}_t - \hat{f}_t}{\hat{\sigma(t)}} \sim \mathcal{N}(0, 1)$.

Here is a statistical summary of the distribution of the standardized residuals in table (6.1.13)

Mean	Standard deviation	Skewness	Kurtozis	Jarque Bera test
0.012	1.00	-0.37	0.61	485

Table 6.1.13: Normality check of standardized residuals

The introduction of this volatility function allows us to get closer to a normal distribution despite the fact that we still reject the null hypothesis of the Jarque Bera test. Note however that if we consider a constant annual volatility function $\hat{\sigma} = \sqrt{8.91}$ and that we test the normality of $u_t = \frac{\tilde{T}_t - \hat{f}_t}{\hat{\sigma}}$ residuals, we obtain the results of table 6.1.14

Mean	Standard deviation	Skewness	Kurtozis	Jarque Bera test
0.01	0.98	-0.40	0.67	565

Table 6.1.14: Normality check of standardized residuals

This shows the improvement brought by the addition of a volatility which depends on the day of the year, allowing us to get closer to a normal distribution in order to respect the assumptions of the model. In order to try to improve this result, we will try to estimate $\hat{\sigma}(t)$ using neural networks.

6.1.6 Neural Networks for detrend volatility

In order to try to get closer to normal standardized residuals we will calibrate neural networks on the deseasonalized volatility computed in the previous section. This new volatility function will be denoted by σ^{NN} and have the following equation

$$\hat{\sigma}(t)^{NN} = \hat{\sigma}(t) + \hat{f}(\sigma_{t-1}, ..., \sigma_{t-p_v}) + \epsilon \tag{6.1.12}$$

where $\hat{f}(\sigma_{t-1}, ..., \sigma_{t-v})$ is a neural network taking as explanatory variables the p_v last volatility values and where $\epsilon_t \sim \mathcal{N}(0, 1)$ and are i.i.d. Note that $\hat{\sigma}(t)$ was computed in section (6.1.5) and allows to recover the series we want to modelize, $\sigma(t) - \hat{\sigma}(t) \approx \hat{f}(\sigma_{t-1}, ..., \sigma_{t-p_v})$. We notice that this series is stationary because it rejects again the null hypothesis of the Dickey Fuller test for several subdivisions of the series. Moreover, we reject the null hypothesis of the Box Ljung test ($p_{value_\alpha} = 0.045$) indicating the presence of a correlation structure in the data. Note however that this hypothesis is weakly rejected, so we do not expect small validation losses given that a large part of this signal is only noise. Finally, we cannot reject the null hypothesis of the Jarque Bera test ($p_{value_\alpha} = 0.089$) leading us to choose a mean square error as loss function as explained in section (6.1.4.2).

The building strategy of these networks will be similar but slightly different from the one presented in section 6.1.4.2. Indeed, we will use shallow network (so only one hidden layer) and we will test a grid containing different numbers of neurons. We will finally choose the one that allows us to minimize our error on the validation set. We will not apply any regularization here in order to avoid the risk of overfitting the validation set. Indeed, as we have only few data at our disposal (365) and as cross validation is not adapted to time series to avoid time leak, we will not use a test set here. We will however take precautions concerning the overfit by using as before a bag fraction in order to test the stability of the network to the input training data. Each of the network will be calibrated 4 times in order to take into account the stochastic side of the networks, as we did the first time. As for section (6.1.4.2), the training data will be shuffled before running in the model in order to promote efficient learning. Each model will receive batches of 4 data and will be trained with the Adam algorithm for 200 epochs. Let us note $p_v = \{3, 5, 7\}$, the set of lag numbers that we test.

Lag 3								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(6,1)	0.8587	0.9092	0.9020	0.9032	0.9346	0.8971	0.0148	31
NN(9,1)	0.7312	0.8909	0.8924	0.8908	0.8901	0.8907	0.0008	46
NN(12,1)	0.7235	0.9177	0.9123	0.9032	0.9161	0.9153	0.00513	61

Table 6.1.15: Training summary Lag 3 volatility models

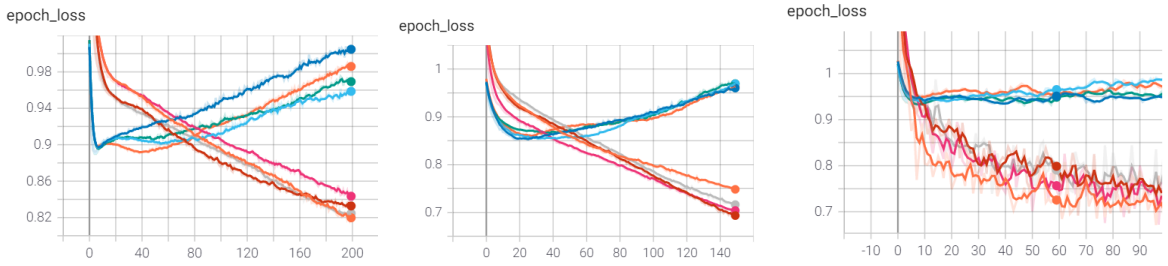
Lag 5								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(8,1)	0.7156	0.8572	0.8522	0.8533	0.8643	0.8588	0.0048	57
NN(12,1)	0.6517	0.9188	0.9119	0.9205	0.9132	0.9295	0.0070	85
NN(16,1)	0.5131	0.9213	0.9072	0.9283	0.9270	0.9225	0 0084	113

Table 6.1.16: Training summary Lag 5 volatility models

Lag 7								
Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation	Parameters
NN(8,1)	0.6063	0.9275	0.9333	0.9292	0.9224	0.9253	0.0041	73
NN(12,1)	0.46215	0.9267	0.9183	0.9239	0.9359	0.9286	0.0064	109
NN(16,1)	0.4022	0.9335	0.9335	0.9365	0.9262	0.9380	0.0045	158

Table 6.1.17: Training summary Lag 7 volatility models

From the results, the optimal value of p_v is 5 and the best model is the NN(8,1) allowing to minimize our validation error. Moreover, this one presents the lowest standard deviation compared to the other architectures for $p_v = 5$ and even if we consider the best model for $p_v = \{5, 7\}$, namely respectively the NN(6,1) and NN(12,1) models. Let us note however that as we had predicted, there is only a small structure to explain, as indicated by the loss values and the result of the Box-Ljung test. As an indication, here are the diagrams related to the training of the best model for each value of p_v.



(a) NN(9,1) (b) NN(8,1) (c) NN(12,1)

Figure 6.1.10: Training and validation losses for the volatility models

The observation of the graphs of the loss functions of each of these models reinforces our decision not to add a regularization. Indeed, the learning process on the validation is not too noisy and allows to avoid an immediate rise of the validation error. Moreover, the validation performance of each model seems stable for each leaf despite the different input data. We will therefore retain the 5 Lags NN(8,1) model. The analysis of the volatility residuals allows us to validate the model (Box Ljung: test $p_{value_{\alpha=0,05}} = 0.20$, JB test: $p_{value_{\alpha=0,05}} = 0.13$), so we fail to reject H_0).

We can now recover the quantity $u_t = \frac{\epsilon_t}{\sigma_t}$ in order to observe its distribution once again for the validation of the hypotheses made in equation (6.1.1). We observe that we unfortunately do not achieve to verify the conditional normality hypothesis. It will therefore be necessary to take this into account in the simulations that will be carried out in the next section.

Mean	Standard deviation	Skewness	Kurtozis	Jarque Bera test
0.012	1.00	-0.37	0.57	458

Table 6.1.18: Normality check of standardized residuals

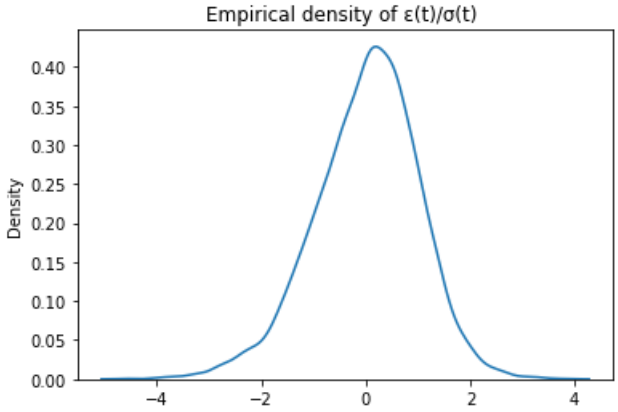


Figure 6.1.11: Empirical standardized residuals distributions

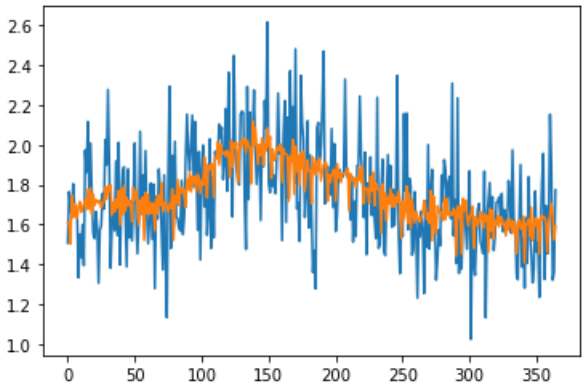


Figure 6.1.12: $\sigma^{N\hat{N}}(t)$ with Neural networks $p_v = 5$

6.1.7 Pricing Method

In order to price our weather derivatives, it is first necessary to be able to reconstruct the indexes on which they are based. This requires the simulation of different temperature trajectories in order to be able to estimate the empirical density of these underlyings. Knowing the different characteristics of the stochastic component of equation (6.1.1), we can establish our simulations. Given the discrete character of our model, the discretization is already fixed to one day so all the previous estimators are based on this step. It is now a question of fixing the statistical properties of the stochastic component $u_t = \frac{\epsilon_t}{\sigma_t}$. In order to test the impact of this hypothesis on the prices of the different products, we will

- Firstly, according to the assumption made in equation (6.1.1), we will assume that $u_t = \frac{\epsilon_t}{\sigma_t} \sim \mathcal{N}(0,1)$. However, we have pointed out in section (6.1.6) that this assumption does not hold.

- Secondly, given that this assumption does not hold, we will simulate trajectories based on the empirical density of standardized residuals, namely the empirical distribution of $u_t = \frac{\epsilon_t}{\sigma_t}$ which we computed in the last section (figure (6.1.11)).

In order to fix the number of simulations necessary for the stability of the temperature density, we will consider that the system is stable when the quantiles $\{0.05, 0.25, 0.5, 0.75, 0995\}$ as well as the mean and the standard deviation do not move anymore during at least 1000 simulations with a precision at the tenth of temperature. We decide to set this requirement at this level since the original data are down to a tenth of a degree Celsius.

First, we will establish the distribution of the various indexes at the maturity date of a one-month contract that would begin on 21/12/2019, the first day of summer. In a second step, we will repeat this analysis but this time for a contract that extends over the first winter month, from 21/06/2020 to 21/07/2020. We will be able to compare the estimates given by the Monte Carlo simulations to the actual temperature at this date since our data extend from 28/11/86 to 30/11/2020. The price of our different financial products will be computed as follows

$$P_t = e^{(-r_t \times (T-t))} \mathbb{EP}(\pi_T) \qquad 0 \leq t \leq T \tag{6.1.13}$$

where P_t is the price at time t of the weather derivative with payoff π_T at maturity and where $e^{(-r_t \times (T-t))}$ is the actualisation factor. For the ease of simplicity, we assume a flat interest rate curve with $r_t = 0 \ \ \forall t \in [0, T]$. This means that, no matter the time and the maturity, the actualisation factor will be equal to one. Moreover, this assumption is in itself credible given the very low interest rate levels we are currently facing. We can then rewrite equation (6.1.13) as

$$P_t = \mathbb{E}^{\mathbb{P}}(\pi_T) \qquad 0 \leq t \leq T \tag{6.1.14}$$

where $\mathbb{E}^{\mathbb{P}}(\pi_T)$ will be estimated by $\mathbb{E}^{\mathbb{P}}(\pi_T) \approx \frac{1}{n} \sum_{t=1}^{m} \pi_t$ where m corresponds to the number of simulations needed to satisfy the convergence constraint which we introduced above. Note that this expectation is computed under the real probability measure P. This will imply the addition of a risk premium to take into account the risk associated with this measure since these products are not priced under a risk neutral measure $\mathbb{Q}$

6.1.7.1 Winter contracts

We will first analyze the results of simulations of different temperature trajectories. You will find in the table below a summary of the distribution of temperatures at the date of the end of the one month duration contract, that is to say at 21/01/2020. From here on, we will note Φ as the empirical density of standardized residuals identified in section (6.1.6).

We observe on table (6.1.20) that the two temperature distributions at maturity are relatively similar, thus indicating that the normality assumption for the standardized residuals seems to provide a good approximation of the empirical distribution for that date. However, we can observe that the empirical distribution has a higher kurtosis [8], which we could have expected given the observations made in figure (6.1.11). Let us add that the temperature actually observed on 21/01/2020 in Madrid was $7.2°C$ indicating that the predictions, which are given by the mean, of the two models are consistent. You can find below the process of convergence of the distributions of table (6.1.20) [9] .

We can now build the corresponding CAT index in order to price weather derivatives on it. Note that the distribution of the CAT index corresponds to the distribution of the sum of the DAT over the duration of the contract. Therefore, it is not because the conditional distributions, with respect of the time, are similar for one day that this will be the case for the distribution of the sum as we see below in table (6.1.22).

[8] In all the pricing section, we mean by kurtozis the "excess kurotsis". This corresponds to the kurtozis - 3.

[9] All the others chart are available in appendix (8.7)

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
$q_{0.005}$	0.356	0.108
$q_{0.25}$	5.774	5.863
$q_{0.50}$	7.719	7.834
$q_{0.75}$	9.582	9.738
$q_{0.995}$	14.995	14.739
$\hat{\mu}$	7.685	7.782
$\hat{\sigma}$	2.804	2.874
Skewness	-0.108	-0.012
Kurtozis	-0.023	0.014
Min	-1.679	-3.538
Max	18.080	17.863
Simulations Needed	5188	6206

Table 6.1.19: Temperature density summary at 21/01/2020

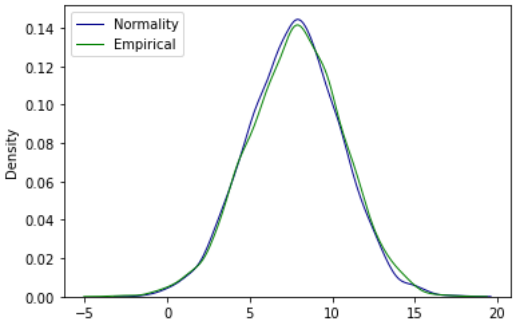
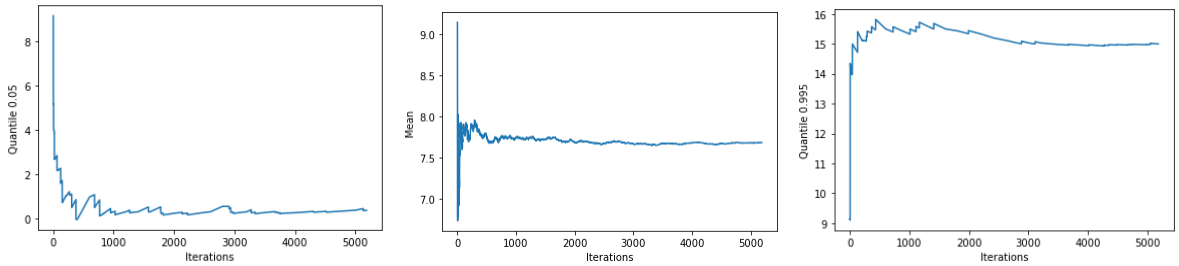


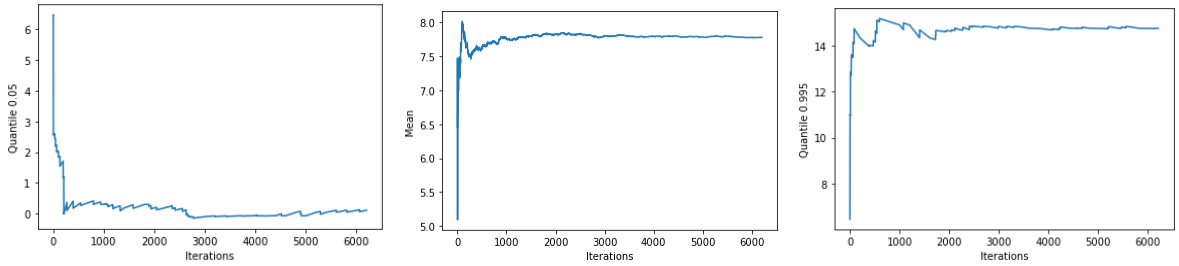
Figure 6.1.13: Temperature density at 21/01/2020

Table 6.1.20: Temperature($°C$) Monte Carlo distribution at 21/01/2020



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 6.1.14: Monte Carlo Distribution with $u_t \sim \mathcal{N}(0,1)$ on 21/12/2020

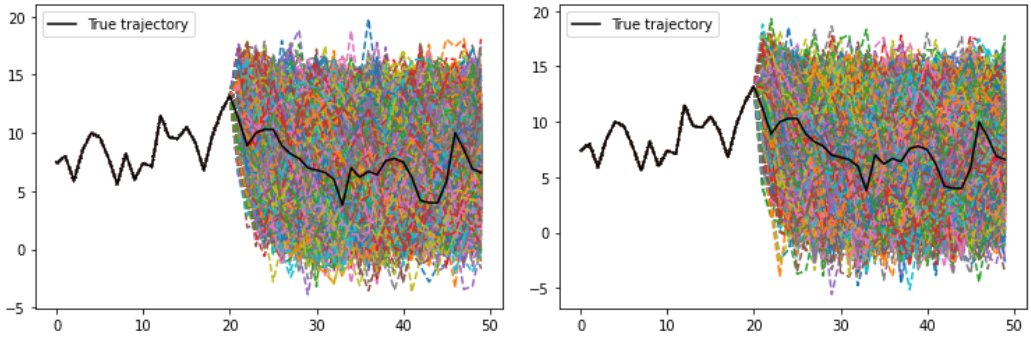


(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 6.1.15: Monte Carlo Distribution with $u_t \sim \Phi$ on 21/12/2020

Such as Zapranis and al. (2009), we observe that the CAT distribution has a higher kurtosis when $u_t \sim \Phi$ and a slightly higher volatility. However, the skewness here is greater (in absolute value) when $u_t \sim \mathcal{N}(0,1)$ but negative in both cases, indicating that the distributions are shifted to the right of the median and thus with tails spread to the left. This indicates that temperatures are more often lower than the median than the reverse, especially in the case where $u_t \sim \mathcal{N}(0,1)$ (range Median-Min = 135.2 , range Max-Median = 111.52). To sum up, when we use the empirical distribution, the values are more extreme but more symmetrical than for the normality assumption, even if it is right skewed for both.

We can establish the price of a futures contract over the period from 21/12/2019 - 21/01/2020. The



(a) Temperatures paths $u_t \sim \mathcal{N}(0,1)$ (b) Temperatures paths with $u_t \sim \Phi$

Figure 6.1.16: Temperatures paths

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
$q_{0.005}$	151.756	157.677
$q_{0.25}$	218.845	220.784
$q_{0.50}$	241.831	245.649
$q_{0.75}$	267.103	270.914
$q_{0.995}$	328.138	334.221
$\hat{\mu}$	242.377	245.830
$\hat{\sigma}$	35.388	35.672
Skewness	-0.031	-0.027
Kurtozis	-0.232	-0.113
Min	106.631	109.518
Max	353.349	356.455
Tail Var (left)	242.898	246.34
Tail Var (right)	336.677	342.125
Simulations Needed	5188	6206

Table 6.1.21: CAT index density summary at 21/01/2020

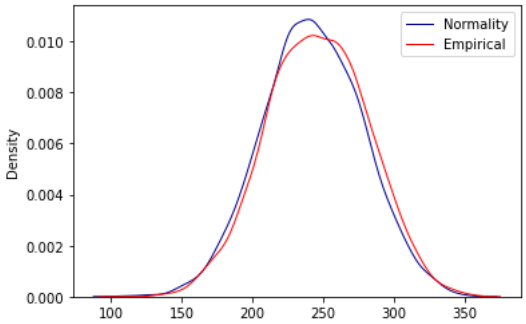


Figure 6.1.17: CAT index density at 21/01/2020

Table 6.1.22: CAT index Monte Carlo density at 21/01/2020

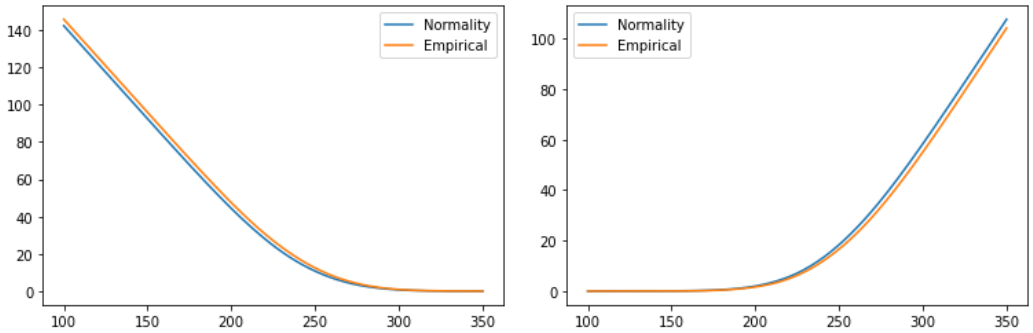
results are in table (6.1.29).

	$[\text{IC}_{95\%}$	$u_t \sim \mathcal{N}(0,1)$	$\text{IC}_{95\%}]$	$[\text{IC}_{95\%}$	$u_t \sim \Phi$	$\text{IC}_{95\%}]$
Futur Price	173.51	242.38	311.53	175.22	245.83	313.75

Table 6.1.23: Futures contract price for 21/12/2019 - 21/12/2020 period

The price of the futures contract under the assumption that $u_t \sim \mathcal{N}(0,1)$ is lower than under the assumption of the empirical distribution. This seems logical at first sight given the observation of the respective kurtozis, skewness and the standard deviations. The study of the right tails of the distributions further indicates that the CAT under the empirical assumption has more extreme scenarios ($q^N_{0.995} < q^E_{0.995}$). In fact, the right Tail Var gives us an indication of the average of the scenarios for which the value of the index exceeds the right Value at Risk ($= q_{0.995}$) indicating on average more extreme scenarios for the case where $u_t \sim \Phi$. We therefore show the risk of using an assumption of marginal normality of the standardized residuals, leading to underpricing the future contract. Note that we have to add a risk premium and if we assume that $u_t \sim \mathcal{N}(0,1)$, we have to take into account the skewness of the distribution if the risk is on the downside (for example a short put option). However,

it would have been accurate to use the empirical distribution for u_t which present a smaller skewness so a more symmetrical risk. In this case, we could think that the risk premium would be an inscreasing function for the volatility. Therefore, the impact of the misspricing will depend of the product and will be heavier for options pricing where the payoff is asymmetric. In fact, if we compare the risk premiums associated to a put and a call option on the CAT index, the difference should be larger for the hypothesis that $u_t \sim \mathcal{N}(0,1)$ to account for the greater skewness. Therefore, we expect to have more expensive call in the case where $u_t \sim \Phi$ and more expensive put when $u_t \sim \mathcal{N}(0,1)$. This is illustrated on figure (6.1.18).



(a) Call options prices on CAT for a range of strikes

(b) Put options prices on CAT index for a range of strikes

Figure 6.1.18: Options prices on CAT index for 21/12/2019 - 21/01/2020 period

We observe well what we expected: the call price when $u_t \sim \Phi$ is higher whatever the strike considered. This is what we could expect by observing the quantiles of the CAT distributions for each of the two hypotheses as well as their moments. Indeed, the shape of the payoff $= (S_T - K)^+$ of a call is in fact the same as truncating the distribution at level K. Therefore, if the distribution of the index under the assumption of normality of the standardized residuals has lower quantiles overall than under the second assumption, we should indeed observe cheaper calls in this case. Given the symmetry of the payoffs for call and puts options, we should logically observe the opposite effect for the prices of put, which is indeed the case. The use of the normal distribution hypothesis leads to underprice calls and overprice put according to our model. Note that this second effect,the overpricing of put, is slightly more important given the introduction of a more negative skewness in the normality case.

6.1.7.2 Summer contracts

Here we will analyze a contract that covers the first month of summer, i.e. from 21/06/2020 to 21/07/2020. As in the previous section, let's start by analyzing the temperature distribution at the maturity date on table (6.1.25).
We again observe that the temperature distributions at maturity differ only slightly under the assumption of marginal distribution of u_t. We observe that the kurtosis with the assumption that $u_t \sim \Phi$ is larger than with the assumption of normal distribution of u_t. Note that the actual temperature at this date was $28.8°C$ where the two models predict respectively $27°C$ and $27.1°C$ on average. You can find below the convergence process of the distributions of the table [10] 6.1.25.
We can again analyze the distribution of the CAT index at the contract maturity date in order to analyze the impact of the standardized residuals distribution assumption in table (6.1.27).
Thanks to these densities, we can establish the price of a futures contract over this period (table (6.1.28)).

The remarks made for the winter contracts are also valid in this case. We observe a left shift of the density as well as a more negative skewness for the CAT index under the assumption of normality of

[10] All the other charts are available in appendix 8.7

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
$q_{0.005}$	19.397	19.110
$q_{0.25}$	25.005	25.129
$q_{0.50}$	26.993	27.180
$q_{0.75}$	28.988	29.151
$q_{0.995}$	34.434	34.384
$\hat{\mu}$	26.990	27.110
$\hat{\sigma}$	2.936	2.983
Skewness	-0.014	-0.121
Kurtozis	-0.089	-0.009
Min	16.82	15.466
Max	37.586	37.361
Simulations Needed	12826	7610

Table 6.1.24: Temperature density summary at 21/07/20

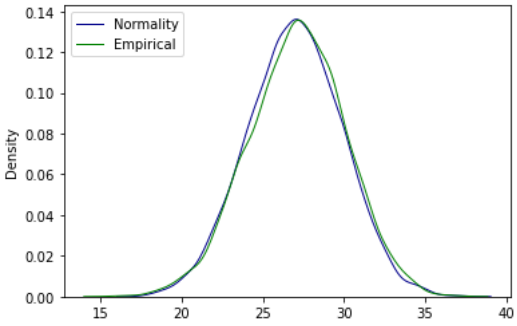
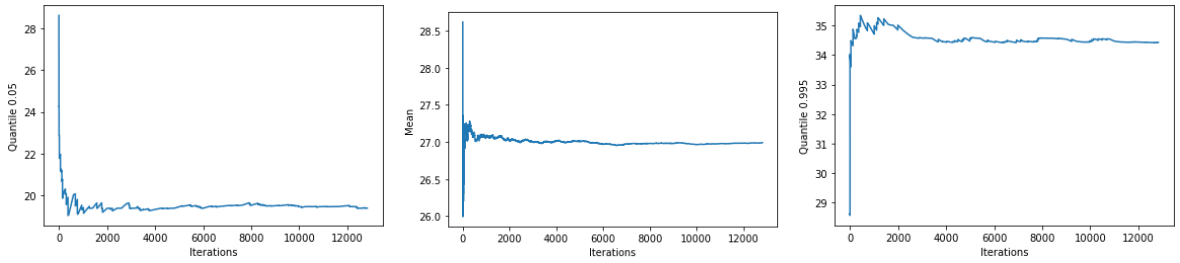


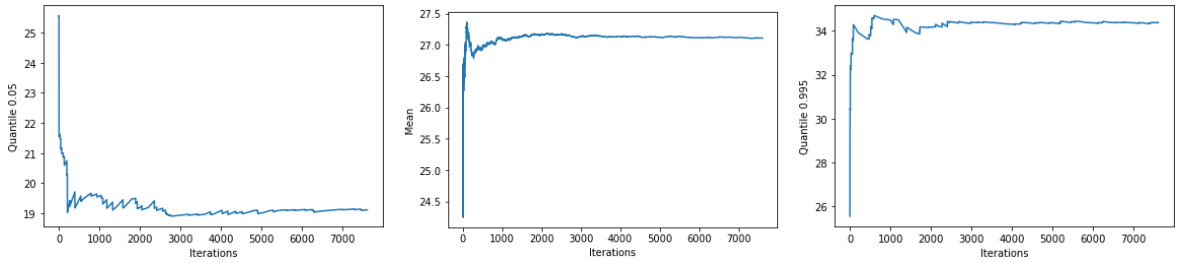
Figure 6.1.19: Temperature density at 21/07/20

Table 6.1.25: Temperature($°C$) Monte Carlo distribution at 21/07/2020



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 6.1.20: Monte Carlo Distribution with $u_t \sim \mathcal{N}(0,1)$ on 21/07/2020



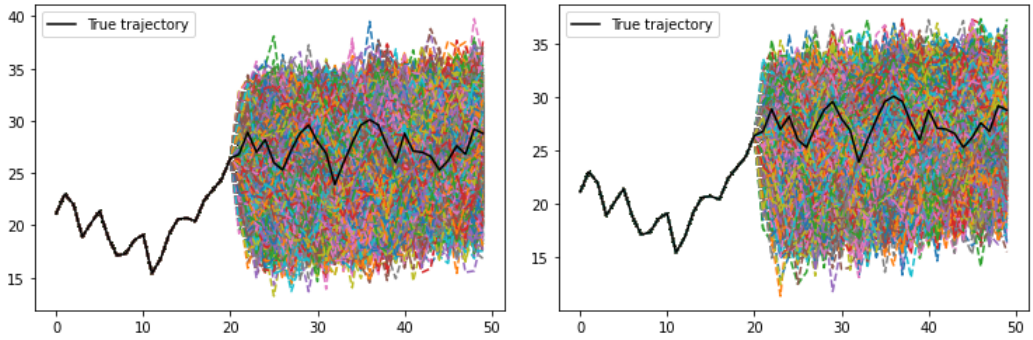
(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 6.1.21: Monte Carlo Distribution with $u_t \sim \Phi$ on 21/07/2020

the standardized residuals, thus giving a lower price for future contract under this assumption. We should therefore observe, as before, expensive call options under the hypothesis that $u_t \sim \Phi$ and the opposite for the case of put options. The figure (6.1.24) confirms this.

We will end this pricing section by pricing futures and options contracts on the CDD index for contracts running from 21/04/2020 to 21/05/2020. We choose this period because the temperatures oscillate around the reference temperature, i.e. $18°C$. Below, the synthesis table (6.1.30) concerning the distributions of this index for each hypothesis of u_t distribution

We observe on table (6.1.30) that the CDD index under the assumption that $u_t \sim \Phi$ has a higher kurtosis and thus leads to more extreme values of the prices (at least in the right part of the distribution),



(a) Temperatures paths with $u_t \sim \mathcal{N}(0,1)$ (b) Temperatures paths with $u_t \sim \Phi$

Figure 6.1.22: Temperatures paths

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
$q_{0.005}$	687.014	693.774
$q_{0.25}$	757.109	759.772
$q_{0.50}$	782.296	785.917
$q_{0.75}$	808.168	812.412
$q_{0.995}$	874.404	877.327
$\hat{\mu}$	782.377	786.067
$\hat{\sigma}$	37.039	37.562
Skewness	-0.029	-0.017
Kurtozis	-0.268	-0.161
Min	641.656	645.987
Max	918.339	905.586
Tail Var (left)	677.312	683.706
Tail Var (right)	881.804	886.156
Simulations Needed	12826	7610

Table 6.1.26: CAT index density summary at 21/07/20

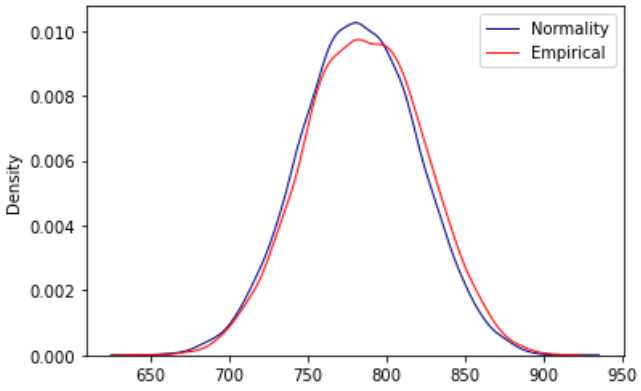


Figure 6.1.23: CAT index density at 21/07/20

Table 6.1.27: CAT index Monte Carlo distribution at 21/07/2020

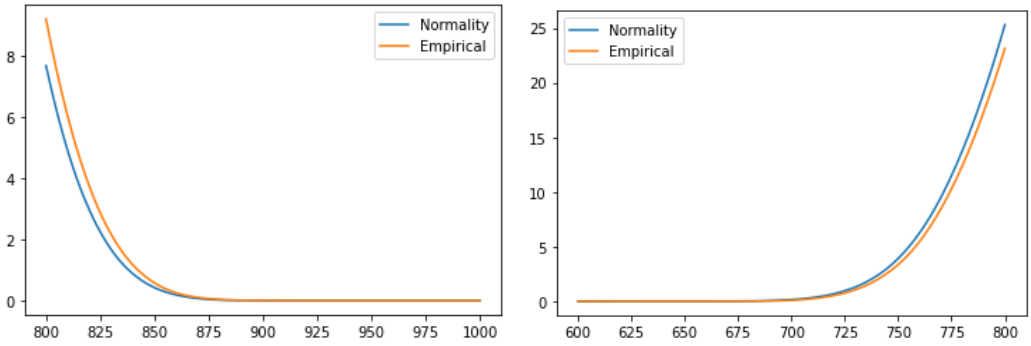
	$[\text{IC}_{95\%}$	$u_t \sim \mathcal{N}(0,1)$	$\text{IC}_{95\%}]$	$[\text{IC}_{95\%}$	$u_t \sim \Phi$	$\text{IC}_{95\%}]$
Futur Price	709.636	782.377	854.125	712.700	786.067	858.651

Table 6.1.28: Futures contract price

which will be calculated on this index. This is further reinforced by the lower skewness of the CDD distribution under the assumption that $u_t \sim \mathcal{N}(0,1)$, highlighting that there are more scenarios where the index takes small values under this assumption, causing the price to decrease. This is in line with what is observed for previous contracts and highlights the risk that is taken when assuming normality of the standardized residuals. This can be seen in the price of potential futures contracts over this period.

We can also compute options prices on the CDD index during this period.

The remarks are similar to those made for the other contracts, the higher kurtozis as well as the higher positive skewness of the CDD density when $u_t \sim \Phi$ explain why the calls are more expensive under this assumption. This also explains the price difference between the put given the symmetry of the payoffs.



(a) CALL price on CAT (b) PUT price on CAT

Figure 6.1.24: Options price on CAT for 21/07/2020 - 21/07/2020 period

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
$q_{0.25}$	11.542	13.41
$q_{0.50}$	22.187	24.285
$q_{0.75}$	36.122	38.077
$q_{0.995}$	81.339	86.203
$\hat{\mu}$	25.389	27.218
$\hat{\sigma}$	17.709	17.855
Skewness	0.853	0.826
Kurtozis	0.419	0.550
Min	0	0
Max	110.292	107.907
Tail Var (right)	87.355	94.544
Simulations Needed	7231	6163

Table 6.1.29: CDD index density summary at 21/07/20

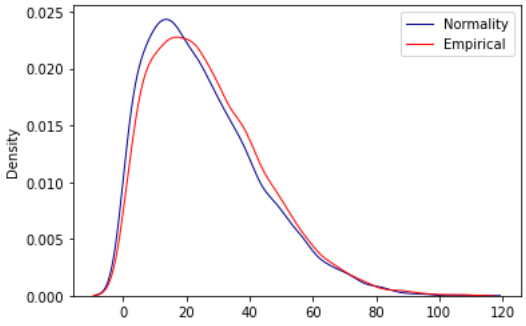
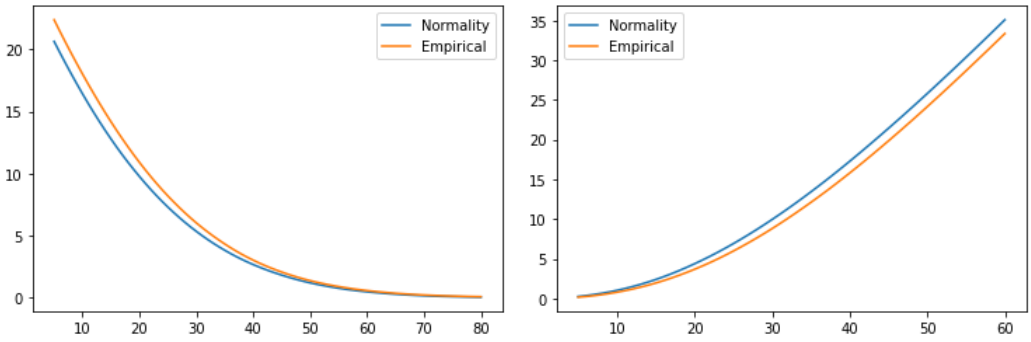


Figure 6.1.25: CDD index density at 21/07/20

Table 6.1.30: CDD indice Monte Carlo distribution at 21/07/2020

	$u_t \sim \mathcal{N}(0,1)$	$u_t \sim \Phi$
Futur Price	25.389	27.218

Table 6.1.31: Futures contract price on CDD Index for the period 21/04/2020 - 21/05/2020



(a) Call options prices on CDD index (b) Put options prices on CDD index

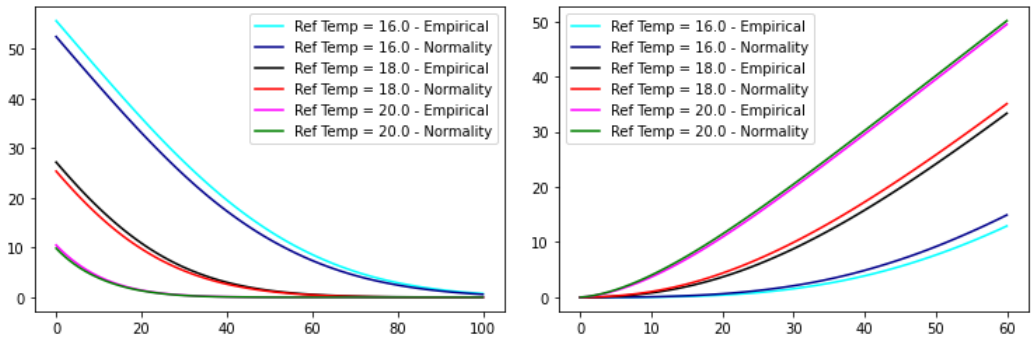
Figure 6.1.26: Options price on CDD for 21/04/2020 - 21/05/2020 period

We can also analyze the sensitivity of this index to the reference temperature T^*. Given the current climate context, it is possible to imagine that this temperature is revised. Therefore, it will be necessary to understand how the prices of futures and options contracts will react to this. The results are in table (6.1.32) for the futures contracts and on figure (6.1.27) for the options contracts.

	$u_t \sim \mathcal{N}(0,1)$	% of change	$u_t \sim \Phi$	% of change
$T^* = 16°C$	52.485	105.93%	55.688	104.60%
$T^* = 18°C$	25.389	100%	27.218	100%
$T^* = 20°C$	9.874	−61.11%	10.523	−61.33%

Table 6.1.32: Futures prices sensitivity on CDD index

Logically, we observe an increase in prices when the reference temperature decreases since the majority of the two distributions lies to the right of this value. The increase is more important for the case where $u_t \sim \mathcal{N}(0,1)$ since the associated distribution is above for values below 18 $°C$ (table (6.1.30)) and explains the change that we can observe in table (6.1.32). When $T^* = 20$, the price decline is similar in both cases. This is justified by the fact that the additional proportion of scenarios where the payoff is zero when the reference temperature rises to 20 $°C$ is similar in both cases. It is indeed when the temperatures are below 18 $°C$ that the two distributions differ the most and where the changes will be the most asymmetric. However, given the current climatic context, if T^* has to be changed, it is more likely to be upwards, limiting the impact of the distribution of u_t. We can now reiterate the analysis but for options.



(a) Call options prices on CDD index (b) Put options prices on CDD index

Figure 6.1.27: Options prices sensitivity on CDD index for 21/04/2020 - 21/05/2020 period

For the options prices, we first observe in figure (6.1.27) that when the reference temperature T^* decreases, the price of calls increases. This is explained by the fact that the CDD index is a decreasing function of T^*. However, if we look at the call prices curves for each of the two hypotheses and for each of the reference temperatures, we observe that these two curves get closer when the reference temperature increases. This phenomenon can be explained by the effect of the greater skewness under the normality hypothesis which makes the two distributions get closer in the right-hand tails. However, given the greater kurtozis of the CDD index under $u_t \sim \Phi$ we always have higher call options prices under the first assumption and the opposite for put options. Note also that if we make T^* tend to zero, we end up with the density of the CAT index over this period. The figure (6.1.28) details the change in density of the CDD index when T^* is modified.

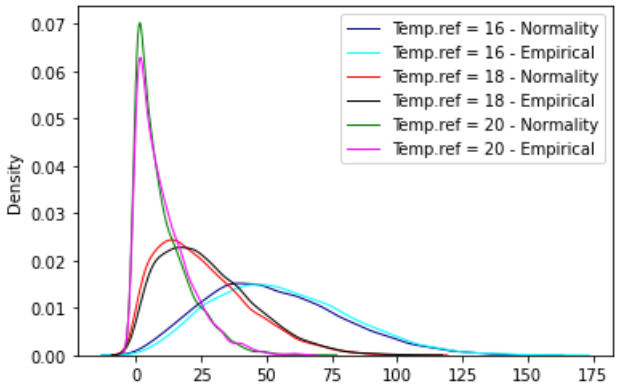


Figure 6.1.28: CDD sensitivity to the reference temperature

We observe that the higher the reference temperature T^*, the less the distributions differ, leading to more similar option prices.

6.2 Multivariate Model

In this section, the methodology is identical to the one adopted in the univariate model (6.1). However, we add here the temperatures of Barcelona in order to see if it allows to refine the predictions of the temperatures of Madrid. In order not to be redundant, this section will have less comments than the previous one, since the methodology is similar. However, we will test the addition of LSTM cells (cfr section 1.3.2.1) to see if they allow a better modeling of temperatures.

6.2.1 Model equation

$$T_t^M = S_t^M + f_t(T_{t-1}^M, ..., T_{t-p}^M, T_{t-1}^B, ..., T_{t-q}^B) + \epsilon^M(t) \tag{6.2.1}$$

$$\tilde{T_t}^M = f_t(T_{t-1}^M, ..., T_{t-p}^M, T_{t-1}^B, ..., T_{t-q}^B) + \epsilon^M(t) \tag{6.2.2}$$

Where the exponents M, B stand for Madrid and Barcelona respectively and where q is the number of lags which we take into account for Madrid temperatures model. Let us note an important detail, if we make the hypothesis that T_t^M is a function of the past temperatures of Barcelona, it will also be necessary to be able to predict the temperatures of Barcelona and thus to build a predictive model for these temperatures. It is however possible in keras to build a multivariate loss function which would allow to calibrate only one model for both time series. However, the goal here is to put forward the potential information gains brought by the temperatures of Barcelona thus requiring a value of loss comparable to the previous ones. The equations (6.2.1) and (6.2.2) are therefore also valid for Barcelona, provided that each of the letters M and B is inverted. Note that each of the elements of the equations (6.2.1) and (6.2.2) are similar to those presented in (6.1.1) and (6.1.2) for the univariate model. The same is true for the assumptions concerning the volatility and noise terms. Recall, however, that we assume that the seasonality term S_t^M as well as the volatility $\sigma^M(t)$ are deterministic functions of the form

$$S_t^M = \underbrace{\lambda^M + \beta^M t}_{\text{Trend}} + \underbrace{\sum_{i=1}^{N_1^M} a_i^M \ \sin\left(\frac{2i\pi(t - f_i^M)}{365}\right) + \sum_{j=1}^{N_2^M} b_j^M \ \cos\left(\frac{2j\pi(t - g_j^M)}{365}\right)}_{\text{seasonality}} \tag{6.2.3}$$

$$\sigma^M(t) = \eta^M + \sum_{i=1}^{J_1^M} c_i^M \ \sin\left(\frac{2i\pi(t - \rho_i^M)}{365}\right) + \sum_{j=1}^{J_2^M} d_j^M \ \cos\left(\frac{2j\pi(t - \gamma_i^M)}{365}\right) \tag{6.2.4}$$

Once again, these equations are also valid for the temperatures in Barcelona provided that the exponent M is changed to B.

6.2.2 Trend and Seasonality calibration

The parameters will be estimated once again by minimizing the square differences between our assumed model for $S_t^{M,B}$ and the corresponding true temperature $y_t^{M,B}$. For the same reasons that have been described previously in the section "Trend and seasonality", it is possible to construct a confidence interval around our different parameters in order to assess their significance level. In order not to be redundant, we will present here only the results for the temperatures of Barcelona since those concerning the city of Madrid are in the table (6.1.1).

Parameters	$[\text{IC}_{95\%}$	Estimate	$\text{IC}_{95\%}]$	Standard deviation
λ^B	15.60	15.68	15.78	4.56×10^{-2}
β^B	4.32×10^{-5}	6.76×10^{-5}	1.10×10^{-4}	6.20×10^{-6}
a_1^B	-7.80	-7.73	-7.67	3.23×10^{-2}
f_1^B	298.92	299.39	299.86	2.42×10^{-1}
a_2^B	0.96	1.02	1.08	3.22×10^{-2}
f_2^B	-0.71	1.08	2.88	9.16×10^{-1}

Table 6.2.1: Trend and seasonal parameters for Barcelona

As for the Madrid case, we found that $N = 2$ is the number of terms required to minimize the least squares. A discrete Fourier transform analysis allows us to observe these two dominant frequencies. We observe however that the confidence interval of the parameter f_2^B contains zero and therefore fails the nullity test for a level $\alpha = 0.05$. We therefore decide to consider that $f_2^B = 0$.

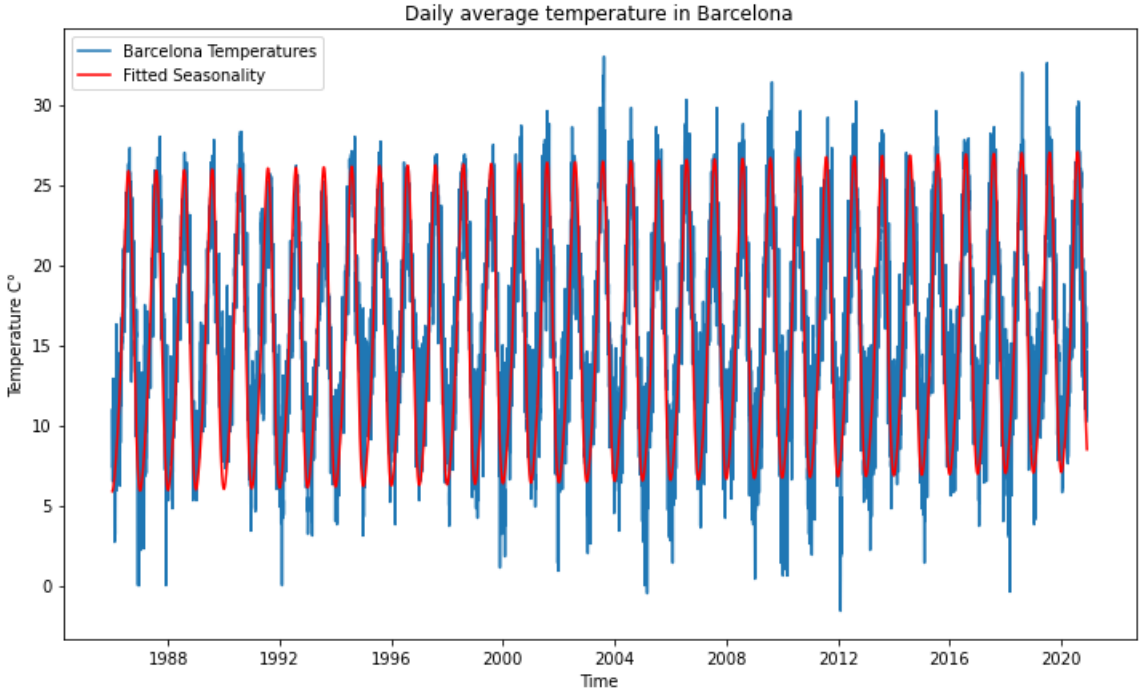


Figure 6.2.1: Fitted trend and seasonality for Barcelona temperatures

6.2.3 Neural Networks modeling

6.2.3.1 Preprocessing

As presented in the section (6.1.4.1), it is first necessary to transform our data so that, it can be processed by our neural networks. We will choose the same standardization as for the univariate model, but we will pay attention to the fact that the temperatures of each of the cities must be standardized independently of the other, i.e. that

$$\begin{aligned} T_j^{*,M} &= \frac{T_j^M - \mu_{T_j^M}^M}{\sigma_{T_j^M}^M} \ \forall j \in \{1, ..., p\} \\ T_j^{*,B} &= \frac{T_j^B - \mu_{T_j^B}^B}{\sigma_{T_j^B}^B} \ \forall j \in \{1, ..., q\} \end{aligned} \tag{6.2.5}$$

if we note $T_j^{M,B}$ the vector of all the j-lagged temperatures of Madrid/Barcelona and where $\mu_{T_j^M}^M$ and $\sigma_{T_j^B}^B$ are respectively the mean and the standard deviation of variable $T_j^{M,B}$.

6.2.3.2 Neural Networks building

The purpose will be to model the temperature of the next day [11], so $\tilde{T}_{t+1}^{M,B}$ according to the temperatures of the last p days $\{\tilde{T}_{t-1}^M, ..., \tilde{T}_{t-p}^M\}$ and the last q days $\{\tilde{T}_{t-1}^B, ..., \tilde{T}_{t-q}^B\}$. We assume here that $p = q = 3$ as we know from the univariate model that 3 lags are the consistent information for the Madrid temperatures. It would still be interesting to test other combinations than $p = q = 3$ but this is left for potential future improvements.

In this section we will test recurrent neural networks in addition to classical ones. The goal will be to compare the validation loss values for each of these architectures in order to determine if the addition of recurrent cells allows a better understanding of the data. The construction strategy of the neural networks (recurrent or not) will be similar to the one presented for the univariate model (cfr 6.1.4.2). Indeed, we will test different combinations of hidden layers and neurons per layer. Moreover, the training data will also be shuffled before being run in the model in order to facilitate convergence during training. We will use the Adam algorithm for weight calibration as it provided the best performance, so we will only report these results. The models will be trained on the basis of a batch of 256 data and for 700 epochs. Note that these models are longer to train because they contain many more parameters as it is possible to see in the section (1.3.2.1). Therefore the training for each of these models was automatically stopped when the validation error did not improve by at least 1×10^{-4} during 10 consecutive epochs or if the 700 epochs were reached. Note that for each architecture, the calibration was stopped before the epochs limit.

[11] For the ease of notation, when building the neural networks, $\tilde{T}^{M,B}$ corresponds to the scaled detrend and deseasonalized temperatures of Madrid or Barcelona.

6.2.3.2.1 LSTM

Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation
LSTM(8,1)	0.3348	0.3535	0.3531	0.3539	0.3540	0.3531	4.288×10^{-4}
LSTM(12,1)	0.3302	0.3522	0.3523	0.3518	0.3521	0.3527	3.223×10^{-4}
LSTM(16,1)	0.3172	0.3521	0.3525	0.3522	0.3523	0.3513	4.610×10^{-4}
LSTM(8,8,1)	0.3160	0.3525	0.3530	0.3524	0.3525	0.3521	2.992×10^{-4}
LSTM(12,12,1)	0.3078	0.3522	0.3523	0.3522	0.3522	0.3520	1.219×10^{-4}
LSTM(16,16,1)	0.3002	0.3533	0.3531	0.3533	0.3535	0.3533	1.615×10^{-4}

Table 6.2.2: LSTM models calibration

We see that all the architectures of the table (6.2.2) allow to improve the value of validation loss compared to the univariate model. However, it is not possible at this stage to know if this improvement is due to the addition of the recurrent cells or simply to the additional information provided by the temperatures of Barcelona. In order to dissociate these effects, we can compare these performances with those of different feed forward networks.

6.2.3.2.2 Feed Forward Networks

Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation
NN(10,1)	0.3356	0.3561	0.3545	0.3548	0.3549	0.3551	1.448×10^{-4}
NN(15,1)	0.3248	0.3553	0.3550	0.3558	0.3559	0.3545	5.811×10^{-4}
NN(20,1)	0.3015	0.3548	0.3527	0.3530	0.3515	0.3518	2.278×10^{-4}
NN(10,10,1)	0.2998	0.3523	0.3560	0.3590	0.3580	0.3593	6.011×10^{-4}
NN(15,15,1)	0.2992	0.3521	0.3518	0.3510	0.3510	0.3516	$3{,}520 \times 10^{-4}$
NN(20,20,1)	0.2897	0.3581	0.3559	0.3562	0.3561	0.3562	1.288×10^{-3}

Table 6.2.3: Feed forward networks models calibration

We thus observe that the improvement in validation loss is mainly related to the addition of Barcelona temperatures. Indeed, the NN(15,15,1) model allows to reach a better validation error than the LSTM(16,1) for a lower standard deviation. The results show that the improvement of the models performance is mainly due to the addition of explanatory variables and not to the addition of LSTM cells. This indicates that the LSTM cells are not necessary to model such small sequences and that the order of arrival of the temperatures does not seem to have an impact on the quality of the predictions. This can be understood by the fact that the detrend and deseasonalized temperatures behave relatively "well" in the sense that they does not change a lot from day to day. Therefore, the samples often have relatively similar values and the order may become less important. Let us note moreover that LSTM models are especially efficient in tasks where the position of the variable in the sequence has a crucial importance such as for text mining tasks. This leads us to the choice of continuing with feed forward networks. We operate in the same way to calibrate different feed forward networks for the temperatures of Barcelona (cfr appendix 8.8). We notice that the validation loss values are higher than for the Madrid temperatures indicating that the series seems less predictable. Indeed, despite different and varied architectures, we did not manage to model the temperatures of Barcelona as we did for Madrid. As explained in the section (6.2.1), the pricing of weather derivatives, in the case of the multivariate model, requires to project also the temperatures of Barcelona, which are as we have just seen, less predictable. This introduces an increase of the model risk which we believe is more important than the potential beneficial impact that these new variables could have on the pricing of weather derivatives. Moreover it seems that the distribution of the detrend and deseasonalized temperatures of Barcelona are relatively far from normal distribution, at least more than in the case of Madrid. It might therefore be useful to change the loss function to take this into account. Note that the NN(15,15,1) achieves a performance of 0.3556 on the test set indicating an improvement compared to the univariate model which achieved a performance of 0.3613 on the same set.

Chapter 7

Conclusion

We were first able to understand in an intuitive but also technical way how a neural network works. We were able to analyze the general idea but also the functioning. This was done for classical architectures, such as feed forward networks, but also for more specific structures such as recurrent networks, more particularly LSTMs. In order to calibrate these models, we also discussed different minimization algorithms that are used in many other fields than machine learning. In this context, we also had to face more general considerations concerning the training of machine learning models leading us to define the notions of overfitting and underfitting as well as the methods allowing to manage them such as regularization or dropout. We were then able to put all of these elements into practice in order to price weather derivatives on different indexes such as the CAT or the CDD and this for different contracts periods along the year. This required pricing using Monte Carlo simulations. These simulations were possible thanks to the modeling of the volatility function on the one hand, and on the other hand thanks to a neural network and the addition of a seasonality to model the temperatures of Madrid. This led us to question the noise term assumption of the initial model and to analyze the impact of its inaccuracy. We can draw several conclusions out of this:

- Concerning neural networks, we note that they require relatively large sizes and therefore a lot of parameters. Therefore it would be relevant to evaluate the real benefits of using a heavier and more "black box" model than a benchmark model. Let us note however that the improvement in performance shows that there is probably a non-linear relationship in the temperatures. This would not be captured by Arma type models because of their too constraining architecture.

- Concerning the modeling of the variance, it would perhaps have been preferable to have more observations so that the estimation of the volatility function would be smoother. However, we were able to show that the normality assumption of the standardized residuals was not verified. This allowed us to analyze the impact of this error on our prices.

- Concerning pricing, we were able to compare the impact of the non-normality of standardized residuals on the prices of different weather derivatives. We found that this assumption introduced too much negative skewness and insufficient kurtosis. This does not have much impact when considering the daily temperatures individually, but it does pose a problem when summing them. Indeed, the indexes (CAT and CDD) have different distributions for each hypothesis and even relatively different prices, indicating a shift in the means. The conditional empirical distribution hypothesis gives more symmetrical indexes and leads to higher prices when the risk exposure is upward, hence the calls. Conversely, the greater negative skewness of the index under the conditional normality assumption leads to higher prices when the risk exposure is down, hence the put options. This erroneous assumption of normality leads us to underprice call options and futures contracts and to overprice put on our different indexes.

- The addition of LSTM cells in the Madrid temperature modeling will not have improved the accuracy of the model. We also tried with bigger sequence, assuming long term dependencies but it does not allow to improve our results.

This thesis opens new ways for future research. It is indeed possible to think of different improvements such as the creation of a loss function more adapted to the modeling of detrend and deseasonalized temperatures. In the idea of reducing the complexity of the model, it would be interesting to study the possible use of a GARCH model to model the volatility function. This could possibly replace the neural network which is more expensive in time and parameters but also less transparent for performances which would be perhaps similar. It would also be interesting to improve the multivariate model. This could be done in different ways. We could first try to improve the modeling of the Barcelona temperatures by trying other types of models. We could also try to add explanatory variables such as precipitation or air pressure. Pricing would however require simulations of each of these variables requiring assessment of the model risk associated with this addition.

Chapter 8

Appendix

8.1 Dummy variables

This method consists in transforming a categorical variable into a data format that can be handled by a machine learning, statistical,... Let's consider a k-level categorical variable. This one can be substituted by $k-1$ binary variables as explained Denuit and al (2019). From then on, each level of this variable will correspond to a unique combination of these $k-1$ variables where each component except one will be equal to zero. This has for cost to increase the space of the explanatory variables which can be a problem when the number of data in the dataset is close to the number of explanatory variables. Let's take the example of a variable "car size" which has the following levels: "big", "medium", "small". This one can be coded as

Explainatory variables	x_1	x_2
Big	0	1
Middle	1	0
Small	0	0

Table 8.1.1: Dummy variables

8.2 Generalized linear model and deviance

Here, we consider n observation vectors $x_i \in \mathbb{R}^{p+1}$ with p explanatory variables and one intercept and where the distribution of ϵ belongs to the exponential dispersion family (Denuit and al. (2019)) with density

$$f_{Y_i}(y) = e^{\left(\frac{y_i\theta_i - a(\theta_i)}{\phi}\right)} c(y,\phi) \tag{8.2.1}$$

where θ_i is the canonical parameter, $a(\theta_i)$ is the cumulant function which is assumed twice continuously differentiable with invertible second derivative, where $\phi > 0$ is the dispersion parameter and where $c(y,\phi)$ doe not depend on θ_i.

Then, the GLM gather all the linear models of the form

$$g(Y_i) = \beta_0 + \beta_1 x_{i,1} + ... + \beta_p x_{i,p} + \epsilon \tag{8.2.2}$$

where Y_i is the response variable, g(.)[1] is the link function and where β is the weights vector. We can then compute its cumulant-moment-generating function (MGF) $\Psi_{Y_i}(t) = \ln \mathbb{E}(e^{tY})$

$$\begin{aligned}\mathbb{E}(e^{tY}) &= \int e^{ty} f_{Y_i}(y|\theta_i, \phi)\ dy \\ &= \int e^{\frac{y(\theta_i+t\phi)-a(\theta_i)}{\phi}} c(y,\phi)\ dy \\ &= e^{\left(\frac{a(\theta_i+t\phi)-a(\theta_i)}{\phi}\right)} \underbrace{\int e^{\frac{y(\theta_i+t\phi)-a(\theta_i+t\phi)}{\phi}} c(y,\phi)\ dy}_{=1}\end{aligned} \tag{8.2.3}$$

We can then compute $\Psi_{Y_i}(t) = \left(\frac{a(\theta_i+t\phi)-a(\theta_i)}{\phi}\right)$ for t close to 0. We can then compute the likelihood function of the ED family. If we consider n observation Y_i that are i.i.d ED distributed, we can write its likelihood function as follows

$$L(y|\theta,\phi) = \prod_{i=1}^{n} e^{\left(\frac{y_i\theta_i - a(\theta_i)}{\phi}\right)} c(_i,\phi) \tag{8.2.4}$$

The log-likelihood function is then

$$\begin{aligned}l(y|\theta,\phi) &= \ln L(y|\theta,\phi) \\ &= \sum_{i=1}^{n}\left(\left(\frac{y_i\theta_i - a(\theta_i)}{\phi}\right)\right) + \sum_{i=1}^{n} \ln c(_i,\phi)\end{aligned} \tag{8.2.5}$$

In order to maximize $l(y|\theta,\phi)$, we have to choose the different β_j such as

$$\frac{\partial l}{\partial \beta_j} = \sum_{i=1}^{n} \frac{y_i - \mu_i}{\phi V(\mu_i) g'(\mu_i)} = 0 \tag{8.2.6}$$

where $\mu_i = \mathbb{E}(Y_i) = a'(\theta_i)$ and where $V(Y_i) = a''(\theta_i)\phi$. We can then introduce the notion of deviance, denoted by D and which is defined as

$$D^*(y|\mu) = 2[l(y|y) - l(y|\mu)] \geq 0 \tag{8.2.7}$$

So minimizing the deviance is the same as maximising, the log-likelihood of Y_i. This makes deviance an ideal candidate to be chosen as a loss function when Y belongs to the ED family to account for the statistical properties of the response variable Y when calibrating the weights β.

8.3 Jarque Bera test

The Jarque Bera statistic aims to test the normality of a series of n data points on the basis of its third and fourth moments, respectively the skewness and the kurtozis, as explained by Abraham and al. (2009). This means that

- H_0 = "The data is normally distributed"
- H_1 = "The data is not normally distributed"

If we denote by the skewness by S and the kurtozis by K, we can write

$$\begin{aligned}S &= \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^3\right] \approx \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^3}{(\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2)^{\frac{3}{2}}} \\ K &= \mathbb{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right] \approx \frac{\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^4}{(\frac{1}{n}\sum_{i=1}^{n}(x_i-\bar{x})^2)^2}\end{aligned} \tag{8.3.1}$$

[1] g(.) has to be at least one time continuous differentiable

where $\frac{X-\mu}{\sigma} \sim \mathcal{N}(0,1)$ under H_0. This test is based on the fact that under H_0, $K_X = 3$ and $S_X = 0$. In order to test the validity of H_0, we can use the JB statistic which is

$$JB = \frac{n-k}{6}\left(S^2 + \frac{(K-3)}{4}\right) \tag{8.3.2}$$

This quantity follows asymptotically a $\mathcal{X}_2^2$ distribution.

8.4 Box-Ljung test

As Abraham (2009) explained, the Box-Ljung Test or the Q statistic is testing the auto correlation in a time series $(X_t)_{t\in\mathbb{N}}$. This means that $H_0 : \rho_1, ..., \rho_{k_max} = 0$ which boils down to verifiy if "There is no serial dependencies left in the data" vs H_1="There is still serial dependencies in the data". In order to test this hypothesis, we can compute

$$Q = n(n+2)\sum_{k=1}^{k_{max}} \frac{\hat{\rho}_k^2}{n-k} \sim \mathcal{X}_{k_{max}}^2 \tag{8.4.1}$$

where n is the sample size, where $\rho_k = Corr(X_t, X_{t-k})$ is the autocrrelation of lag k and where k_{max} is the maximum lag being tested.

8.5 Dickey Fuller test

This test was introduced by Dickey and al (1979). Consider the model

$$Y_t = a + \phi Y_{t-1} + u_t \tag{8.5.1}$$

where u_t is a white noise stochastic process with $\mathbb{E}(u_t) = 0$ and $\mathbb{V}(u_t) = \sigma^2$. Assume Y_0 is fixed, we can rewrite (8.5.1) as

$$Y_t = a(1 + \phi + \phi^2 + ...) + u_t + \phi u_{t-1} + \phi^2 u_{t-2} + ... + \phi^t Y_0 \tag{8.5.2}$$

by substituting iteratively Y_{t-i} by its equation (8.5.1). Then if $a = 0$, corresponding to a random walk without drift, we have different situations depending on the value of ϕ

- $|\ \phi\ | > 1$ all the innovations terms would have a huge impact and the series explodes. This is unrealistic.
- $|\ \phi\ | < 1$ gives a stationary process which is an AR(1).
- $|\ \phi\ | = 1$ gives a random walk. The series is not stationary but $\Delta Y_t = Y_t - Y_{t-1}$ is stationary with mean 0 since

  1. $\mathbb{E}(Y_t) = Y_0$
  2. $\mathbb{V}(Y_t) = \sigma^2 t$

Here, the series fluctuates around a constant with an increasing variance. If we now consider a random walk with drift, so with $a \neq 0$, we have the following case

- $|\ \phi\ | > 1$ all the innovations terms would have a huge impact and the series explodes. This is unrealistic.
- $|\ \phi\ | < 1$ gives a stationary process which is an AR(1).

- $\mid \phi \mid = 1$ gives a random walk. The series is not stationary but $\Delta Y_t = Y_t - Y_{t-1}$ is stationary with mean a since

  1. $\mathbb{E}(Y_t) = Y_0 + at$

  2. $\mathbb{V}(Y_t) = \sigma^2 t$

The series fluctuates around a linear trend $Y_0 + ta$ with increasing variance. Then, we can build the unit root test in order to test the stationary of a time series. We would like to test

- $H_0 : \phi = 1$
- $H_0 : \phi < 1$

in the model described in (8.5.1). As explained above, the series is stationary if $\phi < 1$ so if we reject H_0 meaning that the innovation process $(u_t)_{t\in\mathbb{N}}$ has a transitory effect. The Dickey Fuller test is based on the same idea but the hypotheses are

- $\gamma = 0$
- $\gamma < 0$

This means that it becomes a one-sided t-test for the nullity of the coefficient $\gamma = \phi - 1$ in the test equation

$$\Delta Y_t = a + \gamma Y_{t-1} + u_t \tag{8.5.3}$$

where a, γ are estimated by OLS. Then the test statistic is just

$$t = \frac{\hat{\gamma}}{\sigma_\gamma} \tag{8.5.4}$$

Under H_0 the regressor Y_{t-1} is non stationary and the t statistic described in (8.5.4) does not follow a Student-t distribution but a Dickey Fuller distribution. The critical values of this distribution are more negative than those of normal distribution leading to larger P-values.

8.6 Autoregressive process

As explained Shibata and al (1976), a stationary process Y_t is an autoregressive of order p if it satisfies

$$Y_t = \omega + \sum_{i=1}^{p} \rho_i Y_{t-i} + \epsilon_t \qquad \epsilon_t \sim \mathcal{N}(0, \sigma^2) \tag{8.6.1}$$

where the intercept ω and the vector $\phi = (\phi_1, ..., \phi_p) \in \mathbb{R}^P$ are unknown parameters. These parameters are estimated by maximum likelihood so with a mean square error loss function given that $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$. Under the hypothesis that $|\rho| < 1$ We can compute the expectation and the variance of an AR(1) as follows

$$\begin{aligned} \mathbb{E}(Y_t) &= \mathbb{E}(\omega + \rho \mathbb{Y}_{\approx -\not\Vdash} + \epsilon_{\approx}) \\ &= \omega + \rho \mathbb{E}(Y_{t-1}) \\ &= \frac{\omega}{1-\rho} \end{aligned} \tag{8.6.2}$$

$$\begin{aligned} \mathbb{V}(Y_t) &= \mathbb{V}(\omega + \rho Y_{t-1} + \epsilon_t) \\ &= \rho^2 \mathbb{V}(Y_{t-1}) + \sigma^2 \\ &= \frac{\sigma^2}{1-\rho^2} \end{aligned} \tag{8.6.3}$$

It is also interesting to calculate the autocovariance function

$$\begin{aligned} Cov(Y_t, Y_{t-k}) &= Cov(\rho Y_{t-1} + \epsilon_t, Y_{t-k}) \\ &= \rho Cov(Y_{t-1}, Y_{t-k}) + 0 \end{aligned} \tag{8.6.4}$$

If we note that $\gamma^k = Cov(Y_t, Y_{t-k})$, we can note

$$\begin{aligned} \gamma_k &= \rho \gamma_{k-1} \\ &= \rho^k \gamma_0 \end{aligned} \tag{8.6.5}$$

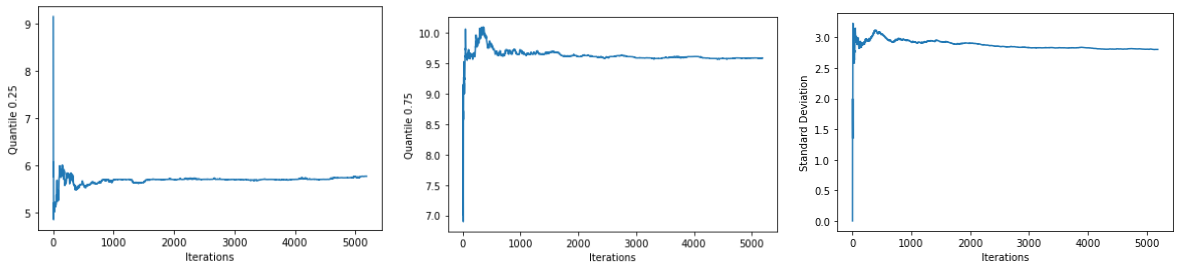
From this result we can compute the autocorrelation function as follows

$$\begin{aligned} Corr(Y_t, Y_{t-k}) &= \frac{Cov(Y_t, Y_{t-k})}{\sqrt{\mathbb{V}(Y_t)\mathbb{V}(Y_{t-k})}} \\ &= \frac{\gamma_k}{\gamma_0} \\ &= \rho^k \end{aligned} \tag{8.6.6}$$

Note that the autocorrelation is equal to zero for all lag greater than p for an AR(p) process. The autocorrelations tend more slowly to zero, and sometimes have a sinusoidal form.

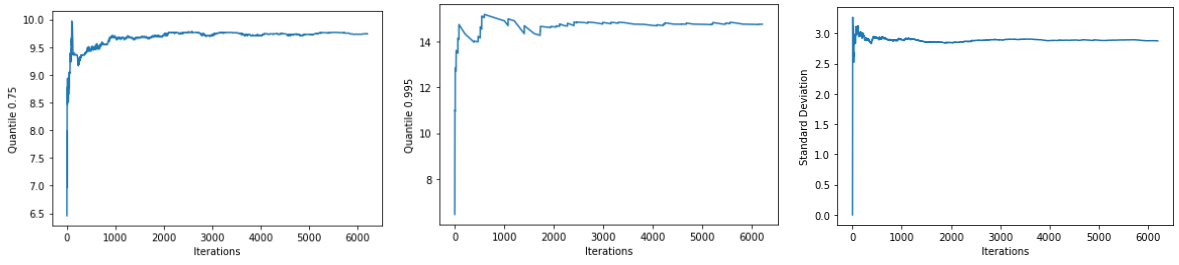
8.7 Convergeance chart

8.7.1 Winter Contract



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

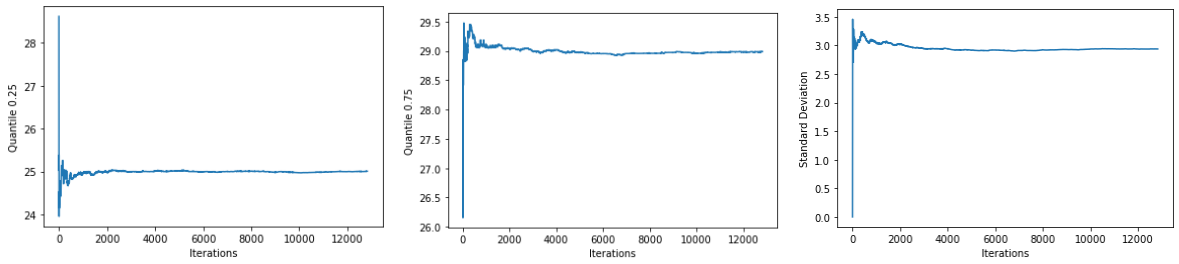
Figure 8.7.1: Monte Carlo Distribution with $u_t \sim \mathcal{N}(0, 1)$ on 21/01/2020



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

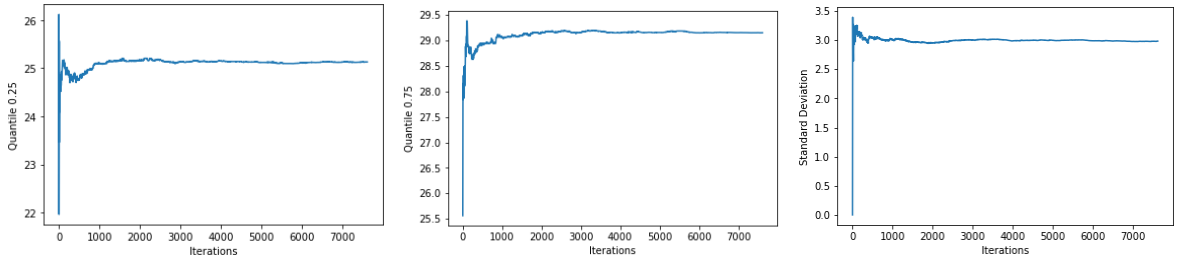
Figure 8.7.2: Monte Carlo Distribution with $u_t \sim \Phi$ on 21/01/2020

8.7.2 Summer Contract



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 8.7.3: Monte Carlo Distribution with $u_t \sim \mathcal{N}(0,1)$ on 21/07/2020



(a) Quantile 0.005 (b) Mean (c) Quantile 0.995

Figure 8.7.4: Monte Carlo Distribution with $u_t \sim \Phi$ on 21/07/2020

8.8 Barcelona temperatures neural networks calibration

Models	Training	Validation	Fold 1	Fold 2	Fold 3	Fold 4	Standard deviation
NN(6,10,1)	0.4836	0.5014	0.5018	0.5016	0.5015	0.5013	1.640×10^{-4}
NN(6,15,1)	0.4729	0.4981	0.4980	0.4980	0.4982	0.4983	1.128×10^{-4}
NN(6,20,1)	0.4701	0.4997	0.4997	0.4995	0.4992	0.4996	2.135×10^{-4}
NN(6,10,10,1)	0.4706	0.4995	0.4988	0.4998	0.4995	0.4999	4.262×10^{-4}
NN(6,15,15,1)	0.4598	0.5013	0.5013	0.5029	0.5004	0.5019	9.243×10^{-4}
NN(6,20,20,1)	0.4512	0.5424	0.5024	0.5038	0.5017	0.4998	1.431×10^{-3}

Table 8.8.1: Feed forward networks models calibration

Bibliography

[1] Bovas Abraham and Johannes Ledolter. *Statistical Methods for Forecasting*. John Wiley & Sons, September 2009. Google-Books-ID: WIPxdb2P8sAC.

[2] Antonis Alexandridis K. and Achilleas D. Zapranis. *Weather Derivatives*. Springer New York, New York, NY, 2013.

[3] Fred ESPEN Benth and Jūratė šaltytė Benth. The volatility of temperature and pricing of weather derivatives. *Quantitative Finance*, 7(5):553–561, October 2007.

[4] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for Hyper-Parameter Optimization. In *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[5] James Bergstra and Yoshua Bengio. Random Search for Hyper-Parameter Optimization. page 25.

[6] Peter J. Bickel, Bo Li, Alexandre B. Tsybakov, Sara A. van de Geer, Bin Yu, Teófilo Valdés, Carlos Rivero, Jianqing Fan, and Aad van der Vaart. Regularization in statistics. *Test*, 15(2):271–344, September 2006.

[7] Léon Bottou. Stochastic Gradient Descent Tricks. In Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, editors, *Neural Networks: Tricks of the Trade*, volume 7700, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[8] Dami Choi, Christopher J. Shallue, Zachary Nado, Jaehoon Lee, Chris J. Maddison, and George E. Dahl. On Empirical Comparisons of Optimizers for Deep Learning. June 2020.

[9] François Chollet and Joseph J. Allaire. *Deep learning with R*. Manning Publications Co, Shelter Island, NY, 2018.

[10] Christine De Mol, Ernesto De Vito, and Lorenzo Rosasco. Elastic-net regularization in learning theory. *Journal of Complexity*, 25(2):201–230, April 2009.

[11] Michel Denuit, Donatien Hainaut, and Julien Trufin. Exponential Dispersion (ED) Distributions. In Michel Denuit, Donatien Hainaut, and Julien Trufin, editors, *Effective Statistical Learning Methods for Actuaries I: GLMs and Extensions*, Springer Actuarial, pages 27–68. Springer International Publishing, Cham, 2019.

[12] Michel Denuit, Donatien Hainaut, and Julien Trufin. Generalized Linear Models (GLMs). In Michel Denuit, Donatien Hainaut, and Julien Trufin, editors, *Effective Statistical Learning Methods for Actuaries I: GLMs and Extensions*, Springer Actuarial, pages 97–196. Springer International Publishing, Cham, 2019.

[13] Michel Denuit, Donatien Hainaut, and Julien Trufin. Maximum Likelihood Estimation. In Michel Denuit, Donatien Hainaut, and Julien Trufin, editors, *Effective Statistical Learning Methods for Actuaries I: GLMs and Extensions*, Springer Actuarial, pages 69–94. Springer International Publishing, Cham, 2019.

[14] David A. Dickey and Wayne A. Fuller. Distribution of the Estimators for Autoregressive Time Series With a Unit Root. *Journal of the American Statistical Association*, 74(366):427–431, 1979. Publisher: [American Statistical Association, Taylor & Francis, Ltd.].

[15] Katharina Eggensperger, Matthias Feurer, Frank Hutter, James Bergstra, Jasper Snoek, Holger H Hoos, and Kevin Leyton-Brown. Towards an Empirical Foundation for Assessing Bayesian Optimization of Hyperparameters. page 5.

[16] Federico Girosi, Michael Jones, and Tomaso Poggio. Regularization Theory and Neural Networks Architectures. *Neural Computation*, 7(2):219–269, March 1995. Conference Name: Neural Computation.

[17] Richard M. Golden and Professor Richard Golden. *Mathematical Methods for Neural Network Analysis and Design*. MIT Press, 1996.

[18] Donatien Hainaut, Michel Denuit, and Julien Truffin. Effective Statistical Learning Methods for Actuaries III: Neural Networks and Extensions | Michel Denuit, Donatien Hainaut, Julien Trufin | download.

[19] Robert Hecht-Nielsen. Theory of the Backpropagation Neural Network. page 13.

[20] Stephen Jewson and Anders Brix. *Weather Derivative Valuation: The Meteorological, Statistical, Financial and Mathematical Foundations*. Cambridge University Press, March 2005.

[21] Chi Jin, Praneeth Netrapalli, and Michael I. Jordan. Accelerated Gradient Descent Escapes Saddle Points Faster than Gradient Descent. In *Conference On Learning Theory*, pages 1042–1085. PMLR, July 2018. ISSN: 2640-3498.

[22] Iebeling Kaastra and Milton Boyd. Designing a neural network for forecasting financial and economic time series. *Neurocomputing*, 10(3):215–236, April 1996.

[23] Sarit Khirirat, Hamid Reza Feyzmahdavian, and Mikael Johansson. Mini-batch gradient descent: Faster convergence under data sparsity. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2880–2887, December 2017.

[24] Donghwan Kim and Jeffrey A. Fessler. Optimized first-order methods for smooth convex minimization. *Mathematical Programming*, 159(1):81–107, September 2016.

[25] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *Artificial Intelligence and Statistics*, pages 528–536. PMLR, April 2017. ISSN: 2640-3498.

[26] Will Koehrsen. A Conceptual Explanation of Bayesian Hyperparameter Optimization for Machine Learning, July 2018.

[27] Simeon Kostadinov. Understanding Backpropagation Algorithm, August 2019.

[28] Thomas Kurbiel. Deriving the Backpropagation Equations from Scratch (Part 1), January 2021.

[29] Friedrich Leisch, Adrian Trapletti, and Kurt Hornik. Stationarity and Stability of Autoregressive Neural Network Processes. In *Advances in Neural Information Processing Systems*, volume 11. MIT Press, 1999.

[30] L. E. Melkumova and S. Ya. Shatskikh. Comparing Ridge and LASSO estimators for data analysis. *Procedia Engineering*, 201:746–755, January 2017.

[31] Jonas Mockus. Application of Bayesian approach to numerical methods of global and stochastic optimization. *Journal of Global Optimization*, 4(4):347–365, June 1994.

[32] Michael A. Nielsen. Neural Networks and Deep Learning. 2015. Publisher: Determination Press.

[33] Emmanuel Okewu, Philip Adewole, and Oladipupo Sennaike. Experimental Comparison of Stochastic Optimizers in Deep Learning. In Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena Stankova, Vladimir Korkhov, Carmelo Torre, Ana Maria A.C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino, editors, *Computational Science and Its Applications – ICCSA 2019*, Lecture Notes in Computer Science, pages 704–715, Cham, 2019. Springer International Publishing.

[34] Alex Sherstinsky. Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. *Physica D: Nonlinear Phenomena*, 404:132306, March 2020. arXiv: 1808.03314.

[35] RITEI SHIBATA. Selection of the order of an autoregressive model by Akaike's information criterion. *Biometrika*, 63(1):117–126, January 1976.

[36] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. page 30.

[37] Divyanshu Thakur. LSTM and its equations, July 2018.

[38] F. Virili and B. Freisleben. Nonstationarity and data preprocessing for neural network predictions of an economic time series. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, volume 5, pages 129–134 vol.5, July 2000. ISSN: 1098-7576.

[39] Bao Wang, Tan M. Nguyen, Andrea L. Bertozzi, Richard G. Baraniuk, and Stanley J. Osher. Scheduled Restart Momentum for Accelerated Stochastic Gradient Descent. *arXiv:2002.10583 [cs, stat]*, April 2020.

[40] Jiazhuo Wang, Jason Xu, and Xuejun Wang. Combination of Hyperband and Bayesian Optimization for Hyperparameter Optimization in Deep Learning. *arXiv:1801.01596 [cs]*, January 2018.

[41] Dongpo Xu, Zhengxue Li, Wei Wu, Xiaoshuai Ding, and Di Qu. Convergence of Gradient Descent Algorithm for a Recurrent Neuron. In Derong Liu, Shumin Fei, Zengguang Hou, Huaguang Zhang, and Changyin Sun, editors, *Advances in Neural Networks – ISNN 2007*, Lecture Notes in Computer Science, pages 117–122, Berlin, Heidelberg, 2007. Springer.

[42] A. Zapranis and A. Alexandridis. Modelling the Temperature Time-dependent Speed of Mean Reversion in the Context of Weather Derivatives Pricing. *Applied Mathematical Finance*, 15(4), August 2008. Publisher: Routledge keywords = Neural networks, weather derivatives pricing, pages = 355–386,.

[43] A. Zapranis and A. Alexandridis. Model Identification in Wavelet Neural Networks Framework. In Iliadis, Maglogiann, Tsoumakasis, Vlahavas, and Bramer, editors, *Artificial Intelligence Applications and Innovations III*, IFIP International Federation for Information Processing, pages 267–276, Boston, MA, 2009. Springer US.

[44] Achilleas Zapranis and Antonis Alexandridis. Modeling and Forecasting CAT and HDD Indices for Weather Derivative Pricing. In Dominic Palmer-Brown, Chrisina Draganova, Elias Pimenidis, and Haris Mouratidis, editors, *Engineering Applications of Neural Networks*, Communications in Computer and Information Science, pages 210–222, Berlin, Heidelberg, 2009. Springer.

[45] Hui Zhong, Zaiyi Chen, Chuan Qin, Zai Huang, Vincent W. Zheng, Tong Xu, and Enhong Chen. Adam revisited: a weighted past gradients perspective. *Frontiers of Computer Science*, 14(5):145309, January 2020.

[46] Martin A. Zinkevich, Alex Smola, Markus Weimer, and Lihong Li. Parallelized stochastic gradient descent. In *in NIPS 23*, pages 2595–2603, 2010.