DEGREE IN ECONOMICS

ACADEMIC YEAR: 2020-2021

END-OF-DEGREE PROJECT

NEURAL NETWORKS AND ARIMA FOR SHORT-TERM ELECTRICITY PRICE FORECASTING.

REDES NEURONALES Y ARIMA PARA LA PREDICCIÓN DEL PRECIO DE LA ELECTRICIDAD A CORTO PLAZO.

AUTHOR:
ADRIÁN GONZÁLEZ CARPINTERO

DIRECTOR:
JOSÉ LUIS GALLEGO GÓMEZ

June 2021

Abstract

Electricity price forecasting provides significant information for the different electricity market agents so that their profits can be maximized. This work is meant to make a univariate and multivariate comparison between state-of-the-art statistical models such as ARIMA and Transfer Function Models, and the promising Deep Learning models, such as Recurrent Neural Networks and Convolutional Neural Networks, in order to make 24 hours ahead predictions of the electricity price in the Spanish electricity market for the 2020 timespan. In addition, an ensembling model composed of models from both backgrounds will be suggested to improve the predictions of either individual model. In the experiments, Convolutional Neural Networks outperformed all other Neural Networks at univariate and multivariate level and had similar results to the state-of-the-art statistical models at univariate level, outperforming them at multivariate level. Additionally, it has been shown that the ensembling model obtains considerably better results than each of the individual models.

Contents

1 INTRODUCTION **4**

2 POWER MARKET AND PRICING **5**

3 ARIMA MODELS **6**
3.1 SEASONAL PROCESSES 7
3.2 TRANSFER FUNCTION MODELS 7

4 NEURAL NETWORK MODELS **8**
4.1 ARTIFICIAL NEURAL NETWORKS 8
4.2 RECURRENT NEURAL NETWORKS 11
4.2.1 Long Short Term Memory 13
4.2.2 Gated Recurrent Units 14
4.3 CONVOLUTIONAL NEURAL NETWORKS 14

5 DATA ANALYSIS **17**
5.1 DATABASE 17
5.2 EXOGENOUS VARIABLE SELECTION 17
5.3 DESCRIPTIVE ANALYSIS 18
5.3.1 Hourly behavior 18
5.3.2 Daily behavior 19
5.3.3 Seasonal behavior 19
5.4 OUTLIER ANALYSIS 19
5.4.1 Explanation of some outliers 20

6 ARIMA MODELS PROPOSED FOR FORECASTING **24**
6.1 IDENTIFICATION 24
6.1.1 Stationarity 24
6.1.2 Finding ARMA structure 26
6.2 ESTIMATION AND DIAGNOSTIC CHECKING 26
6.3 TRANSFER FUNCTION MODELING 28

7 NEURAL NETWORKS MODELS PROPOSED FOR FORECASTING **30**
7.1 ENVIRONMENT FOR SUPERVISED LEARNING 30
7.2 NEURAL NETWORKS OPTIMIZATION 31
7.2.1 Choosing the strategy for multi-step ahead forecasting 31
7.2.2 Lags selection 31
7.2.3 Exogenous variables 31
7.2.4 Data normalization 31
7.2.5 Neurons activation function 32
7.2.6 Learning rate schedule 32
7.2.7 L2 regularization 33
7.2.8 Retraining 33
7.2.9 Ensembling 33
7.2.10 Hyperparameters optimization 33
7.3 NEURAL NETWORKS ARCHITECTURES PROPOSED 33
7.3.1 Convolutional Neural Network 34
7.3.2 Recurrent Neural Network 34
7.3.3 Hybrid CNN-RNN model 34

8 NEURAL NETWORKS AND ARIMA ENSEMBLING **34**

9 RESULTS **35**
9.1 UNIVARIATE MODELS RESULTS 35
9.1.1 Detailed analysis of the results 36
9.2 MULTIVARIATE MODELS RESULTS 38
9.3 ARIMA AND CNN ENSEMBLING 39

10 CONCLUSION **41**

References **43**

A Python code for power market and pricing section **44**

B Code for data analysis section **44**
B.1 Python code for correlation matrix 44
B.2 R Code for descriptive analysis 45
B.3 Python code for the Encoder-Decoder 47
B.4 Python code for outliers graphs 49
B.4.1 Code for reconstructed time series and outliers graphs 49
B.4.2 Code for supply and demand graphs 50

C R code for ARIMA modeling **52**
C.1 R code for ARIMA model 52
C.2 R code for Transfer Function Model 53

D Python code for Neural Networks modeling **56**
D.1 Python code for Recursive Multi-step Neural Network 56
D.2 Python code for hyperparameter optimization with keras tuner 58
D.3 Python code for Neural Networks models (CNN, RNN and CNN-RNN) . . . 59

E Neural Networks architectures diagrams (for univariate models) **63**
E.1 CNN diagram 63
E.2 RNN diagram 63
E.3 CNN-RNN diagram 64

F Python code for ARIMA and NN ensembling **64**

G Python code for results section **65**
G.1 Python code for Naive model 65
G.2 Python code for tables and barplots 65
G.3 Python code for kernels graph 68

1 INTRODUCTION

ARIMA models have been part of the state-of-the-art in time series forecasting since the popularization of the Box-Jenkins methodology in the 1970s. Regarding Neural Networks (NN), although first NN was developed in 1958, it is not until 2006, with the emergence of Deep Learning (DL) and advances in computational potential, that they began to gain special attention, achieving an outstanding performance on many important problems in computer vision, speech recognition, and natural language processing (Gu et al. 2018).

In order to find the state of the art for time series prediction, Spyros Makridakis established in 1982 the M-competition, where 15 forecasting methods were tested for 1001 time series (Makridakis et al. 1982). From then on, the competition gained popularity, giving birth to M2 in 1993, M3 in 2000, M4 in 2018 and M5 in 2020, and becoming an institution of great importance to know the progress of the different forecasting methods. In M1, M2 and M3 the methods tested were purely statistical, and it is not until M4, when due to the growing interest in the area, Machine Learning methods were introduced. However, these methods turned out not to be up to the task, obtaining general poor performances (Makridakis et al. 2018).

In 2020 the M5 is launched, being the largest competition to date and involving more than 7000 participants on developing forecasting methods for a retail sales forecasting application. Unlike the M4, Machine Learning methods obtained excellent performance, leading to a disruptive result never seen before in the history of the competitions (Makridakis et al. 2020). In particular, LightGBM, an algorithm based on decision trees, was the best performer, forming part of most of the models in the top 50. NN were also among the first results, such as the second place, which uses N-BEATS in addition to LightGBM, or the third place, which uses a model composed entirely of 43 NN. On the performance of DL, Makridakis concludes: “deep learning methods like DeepAR and N-BEATS, providing advanced, state-of-the-art ML implementations, have shown forecasting potential motivating further research in this direction”. These statements are supported by the continuous advances of DL over the last decade (Wani et al. 2020) which, supported by the flexibility of the model, suggest a promising future for these methods in the field of time series prediction.

Regarding the electricity market, it is worth mentioning that making short-term predictions can be profitable for different agents: On the supply side, energy supply companies that have freedom to vary the quantity supplied, as dam-type hydroelectric, natural gas, and fuel oil power plants. On the demand side, companies could save costs (specially if they are power usage intensive) if they were flexible enough to schedule their operations at different hours throughout the day.

Zareipour et al. (2009) made a quantification of the amount saved by two companies with different characteristics: a process industry owning on-site generation facilities and a municipal water plant with load-shifting capabilities. For the process industry they found the following: “when MAPE falls between 6 to 16%, a 1% improvement in MAPE would result in about 0.2% reduction in operation costs; this reduction is higher when the forecast MAPE falls within the 16 to 24% range”, and for the water plant: “when MAPE is within a 5 to 14% range, a 1% improvement in forecast accuracy would result in about 0.35% cost reductions for this case study. Beyond this range, the rate of reduction is higher”.

In this work, both ARIMA and NN models will be developed in order to predict 24 hours ahead prices for 2020 timespan. For this purpose, a theoretical explanation of each model will be given first, followed by a data analysis section to know the different features of the series. After this, we will proceed with the specification of each model, and the results will be discussed in the last section.

ARIMA models have been developed in R using `tfarima` package, which allows to describe series with multiple cycles and seasonalities, while NN models have been developed in Python with `Tensorflow` and `Keras` libraries, which allows the design of customized NN prototypes according to the analyst's needs. All the code used for the elaboration of this work can be found in the Appendix and in my GitHub.

2 POWER MARKET AND PRICING

The Spanish electricity market is made up of a series of markets: the forward market, the daily market and the intraday market, in addition to a series of operations such as bilateral contracts or private agreements between companies. Market management is carried out by OMIE (Operador del Mercado Ibérico de Energía). As the purpose of this work is to make predictions for the daily market price, where most of the energy is traded, the aim of this section is to understand what are the main determinants of this price.

The bid matching process is carried out by an algorithm called Euphemia, which comes from the Price Coupling of Regions initiative and operates in all Europe. Its purpose is to compute the price of electric energy in order to maximize the surplus of buyers and sellers. For this purpose, we first need to understand how the power market is build. In order to build the supply curve, we need to know each of the supplies (energy quantity and price) of each of the companies in charge of energy generation, and in order to build the demand curve, we need to know each of the demands (energy quantity and price) of each market agent. Buyers in the energy production market are mainly traders and direct consumers (OMIE 2020).

Once the purchase and sale offers have been made, they are sorted by price, in increasing order in the case of supply and decreasing order in the case of demand. The point where the curves intersect will represent the matching point, where the total surplus is maximized. This point will determine the amount of energy to be marketed and its price. This process is repeated for the 24 hours of the following day and takes place every day at 12:00 CET for all Europe.

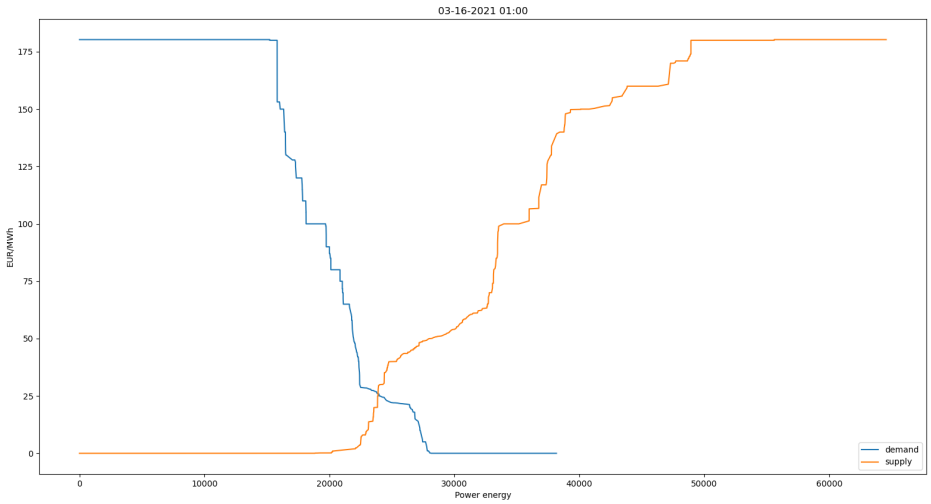


Figure 1: Demand and supply for an hour of the day. Code in Appendix A.

However, the determination of supply and demand curves also presents a series of complexities that cause changes in these curves and therefore in pricing. The electric power sale offers may incorporate complex conditions, which are grouped into 4 groups: indivisibility condition, load gradient, minimum revenue condition and scheduled outage condition (OMIE 2020). In addition, there is the possibility of importing and exporting to bordering countries (France, Portugal, Morocco and Andorra), which also causes changes in the supply and demand curve. All of this complexities are the reason why when presenting supply and demand curves, OMIE differentiates between demand and matched demand, and supply and matched supply.

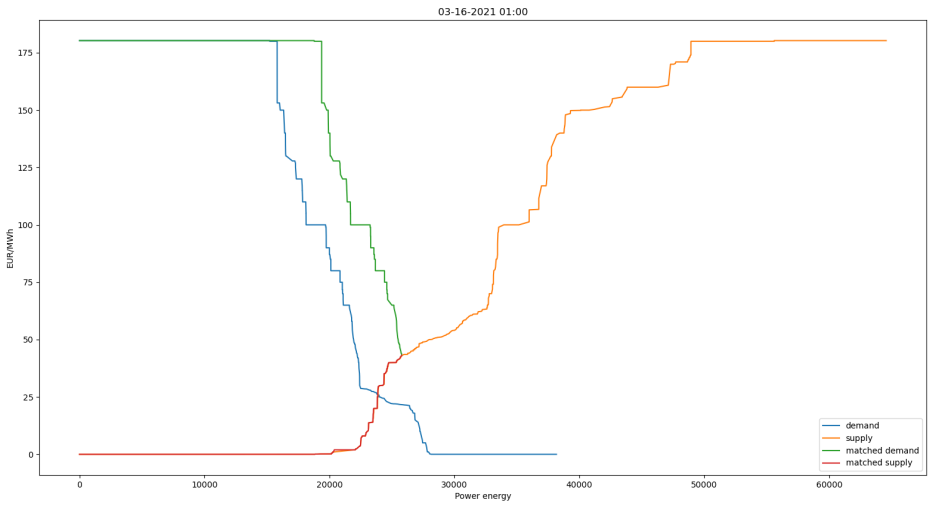


Figure 2: Demand, supply, matched demand and matched supply for an hour of the day. Code in Appendix A.

As wee see, pricing comes as a consequence of numerous complexities. In data analysis section we will go deeper into this.

3 ARIMA MODELS

Box and Jenkins popularized the autorregresive integrated moving average process, ARIMA(p,d,q) in the early 1970s (Box & Jenkins 1976). This process consists of an autorregresive operator AR, a moving average operator MA, and an integrated operator I:

$$\phi_p(B)\nabla^d Y_t = \delta + \theta_q(B)u_t \tag{1}$$

where p is the autorregresive order, d indicates the number of differences, and q is the moving average order. $\phi_p(B)$ is the simplified form of the AR operator $1-\phi_1 B-\cdots-\phi_p B^p$, $\theta_q(B)$ is the simplified form of the MA operator $1-\theta_1 B-\cdots-\theta_q B^q$, ∇^d is the difference operator defined as $\nabla^d = 1-B^d$ and B is the backward shift operator.

Box-Jenkins methodology is structured in three stages: identification, which aims to suggest a tentative model, estimation, where the parameters suggested are estimated,

and diagnosis checking, where different tools are used in order to ascertain whether the model is appropriate or not. Our main tools through this process will be the autocorrelation function (ACF),

$$cor(Y_t, Y_{t-k}) = \frac{cov(Y_t, Y_{t-k})}{\sqrt{V(Y_t)V(Y_{t-k})}} = \frac{\gamma_k}{\gamma_0} = \rho_k \tag{2}$$

and the partial autocorrelation function (PACF),

$$cor(Y_t, Y_{t-k}|Y_{t-1}, \dots, Y_{t-k+1}) = \phi_{kk} \tag{3}$$

3.1 SEASONAL PROCESSES

In order to deal with seasonality we use the process ARIMA(P,D,Q)$_S$:

$$\Phi_P(B^S)\nabla_S^D Y_t = \Theta_Q(B^S)u_t \tag{4}$$

Where P is the order of the seasonal autorregresive polynomial, D the number of seasonal differences and Q the order of the seasonal moving average polynomial. Joining the regular structure with the seasonal we can write the multiplicative ARIMA(p,d,q)(P,D,Q)$_S$ process as follows:

$$\phi_p(B)\Phi_P(B^S)\nabla^d\nabla_S^D Y_t = \theta_q(B)\Theta_Q(B^S)u_t \tag{5}$$

3.2 TRANSFER FUNCTION MODELS

In transfer function models we relate two or more time series in order to make predictions. In this way, we will have an input, X, and an output Y, related via the following equation:

$$Y_t = \upsilon(B)X_t + N_t \tag{6}$$

where $\upsilon(B) = (\upsilon_0 + \upsilon_1 B + \upsilon_2 B^2 + \dots)$ and N_t is the error term, independent of the input variable. As $\upsilon(B)$ has infinite parameters, we can rewrite it as follows:

$$\upsilon(B) = \frac{\omega_s(B)B^b}{\delta_r(B)} \tag{7}$$

where $\omega(B) = \omega_0 - \omega_1 B - \omega_2 B^2 - \dots - \omega_s B^s$ and $\delta(B) = \delta_0 - \delta_1 B - \delta_2 B^2 - \dots - \delta_r B^r$.

On the other hand, we can make the error term N_t follow an ARIMA process as the one defined in the previous section:

$$N_t = \frac{\theta_q(B)}{\phi_p(B)(1-B)^d}u_t \tag{8}$$

Thus, if we define y_t and x_t as stationary variables: $y_t = (1-B)^d Y_t$, $x_t = (1-B)^d X_t$
We can rewrite (6) as follows:

$$y_t = \frac{\omega_s(B)B^b}{\delta_r(B)}x_t + \frac{\theta_q(B)}{\phi_p(B)}u_t \tag{9}$$

This equation can be adapted to a number k of inputs in the following way:

$$y_t = \sum_{i=1}^{k}\frac{\omega_{s_i}(B)B^{b_i}}{\delta_{r_i}(B)}x_{it} + \frac{\theta_q(B)}{\phi_p(B)}u_t \tag{10}$$

And also include seasonal parameters:

$$y_t = \sum_{i=1}^{k} \frac{\omega_{s_i}(B)\Omega_{S_i}(B^{S_i})B^{b_i}}{\delta_{r_i}(B)\Delta_{R_i}(B^{S_i})} x_{it} + \frac{\theta_q(B)\Theta_Q(B^S)}{\phi_p(B)\Phi_P(B^S)} u_t \tag{11}$$

Methodology for the Transfer Function models follows the same stages as for the ARIMA model (identification, estimation and diagnosis checking). Our main tool for identifying s, b and r will be the Cross-correlation function (CCF),

$$\rho_{yx}(j) = \frac{\gamma_{yx}(j)}{\sigma_y \sigma_x} \tag{12}$$

where $\gamma_{yx}(j)$ is the Cross-covariance,

$$\gamma_{yx}(j) = E[(y_t - \mu_y)(x_{t-j} - \mu_x)] \qquad j = 0, \pm 1, \pm 2, \ldots \tag{13}$$

4 NEURAL NETWORK MODELS

4.1 ARTIFICIAL NEURAL NETWORKS

Artificial Neural Networks (ANN) are a supervised learning algorithm within in a branch of artificial intelligence called Machine Learning. The algorithm, inspired by biological neural networks, is made up by layers of neurons that process an input to produce an output (prediction). First NN, the perceptron, was proposed by Frank Rosenblatt in 1958 (Rosenblatt 1958), and consisted of one neuron and one layer:

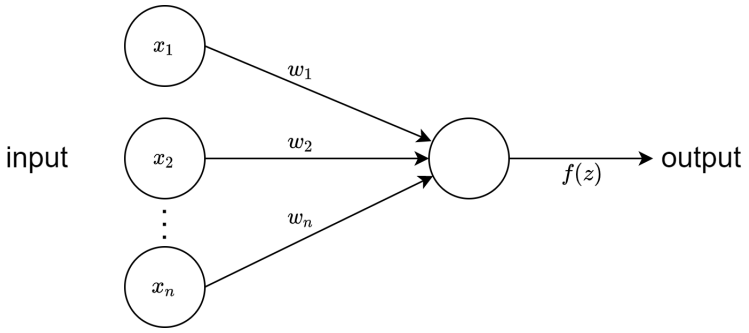


Figure 3: Diagram of a perceptron.

Input is defined by a vector $X = (x_1, x_2, ..., x_n)$. Rosenblatt also defined the concept of weights, real numbers with the purpose of matching the inputs with the outputs. For the perceptron, a weight vector is paired up with the input vector via linear combination, and then an activation function, $f(z)$, is passed to it. A bias (b) is occasionally included, resulting in the following expression:

$$a = f\left(\sum_j w_j x_j + b\right) \tag{14}$$

where a is the output. The single perceptron can be used as a building block of more complex structures such as the multilayer perceptron (MLP): a ANN with multiple layers and neurons where each layer is connected with his contiguous via a weight matrix.

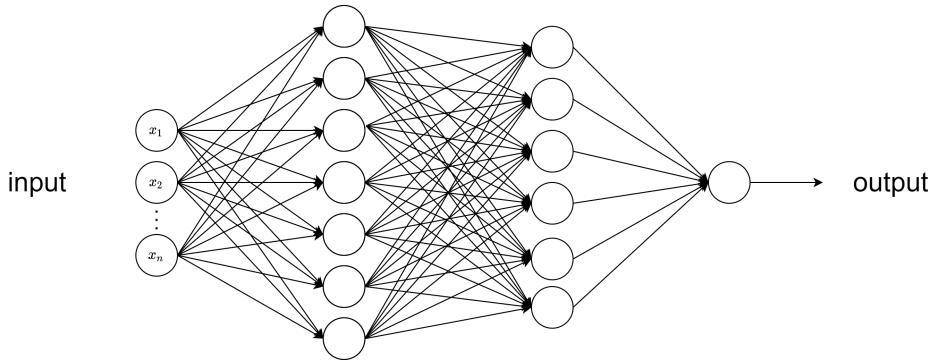


Figure 4: Multilayer perceptron.

The used notation is the following: l represents the layer, j the neuron in the layer l and k the neuron in the layer $l-1$. In this way the activation a_j^l from the j neuron of the l layer will be:

$$a_j^l = f\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{15}$$

In matrix-vector notation, if we define w^l as the weight matrix from the l layer, b^l as the bias vector from the l layer and a^l as the activation vector from the l layer, we have:

$$a^l = f\left(w^l a^{l-1} + b^l\right) \tag{16}$$

With this simple formula we can summarize the way a NN is fed, i.e, how it generates predictions from inputs.

In supervised learning the goal is to make the model learn from labeled data, i.e, data where the input and the output are known in order to compare the predictions made by the algorithm with the real values. Therefore, we need to split our database into two sets: input and output data. Then we apply the following method: as for every input we have a prediction and an output, we will define a cost function, i.e., a function that captures the differences between the predictions and the real values. We will compute the cost function and then update weights and biases in order to minimize it. This method is carried out with a chunk of the data called training data. The rest of the data, called test set, will be taken in order to evaluate the model after the training, thus we can see how good the NN is at making predictions with new data.

In order to minimize the cost function we use a tool called gradient descent. For the cost function $C(v)$ where v is a vector, we define the gradient of C, ∇C, as the vector of partial derivatives:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}, ..., \frac{\partial C}{\partial v_n}\right) \tag{17}$$

Knowing that $\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_1}\Delta v_2 + \ ... \ + \frac{\partial C}{\partial v_1}\Delta v_n$ we can rewrite:

$$\Delta C \approx \nabla C \cdot \Delta v \tag{18}$$

If we choose Δv in such a way that ΔC is negative, we can write:

$$\Delta v = -\eta \nabla C \tag{19}$$

Where η is a parameter called learning rate.

Knowing this, the way weights and biases will be updated is the following:

$$w_{jk}^{l} \rightarrow w_{jk}^{l\prime} = w_{jk}^{l} - \eta \frac{\partial C}{\partial w_{jk}^{l}} \tag{20}$$

$$b_{j}^{l} \rightarrow b_{j}^{l\prime} = b_{j}^{l} - \eta \frac{\partial C}{\partial b_{j}^{l}} \tag{21}$$

Furthermore, in order to accelerate the learning process it is common to use a technique called stochastic gradient descent. This technique uses the concept of mini-batches: groups of training inputs composed by m random inputs: $X_1, X_2, ..., X_m$. We can write:

$$\nabla C \approx \frac{1}{m} \sum_{i=1}^{m} \nabla C_{X_i} \tag{22}$$

By doing so we can express (20) and (21) as:

$$w_{jk}^{l} \rightarrow w_{jk}^{l\prime} = w_{jk}^{l} - \frac{\eta}{m} \sum_{i}^{m} \frac{\partial C_{X_i}}{\partial w_{jk}^{l}} \tag{23}$$

$$b_{j}^{l} \rightarrow b_{j}^{l\prime} = b_{j}^{l} - \frac{\eta}{m} \sum_{i}^{m} \frac{\partial C_{X_i}}{\partial b_{j}^{l}} \tag{24}$$

That being said, we need a method to compute the gradient. Here is where the learning engine of the ANN appears: the backpropagation algorithm. In order to apply backpropagation we need to make two assumptions (Nielsen 2015):

1. That the cost function can be written as a mean of cost functions for individual training samples, that is: $C = \frac{1}{n} \sum_x C_x$.
2. That the cost can be written as a function of the NN outputs, that is, $C = C(a^L)$.

We also need to define the following: from (16), if we eliminate the activation function, we have:

$$z^l = \left(w^l a^{l-1} + b^l \right) \tag{25}$$

Hence $a^l = f(z^l)$. We define δ_j^l as the error in the j neuron from the l layer:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{26}$$

Therefore δ^l will be the vector of errors associated with l layer. The goal of backpropagation is to compute δ^l for each layer, and from that compute the partial derivatives of interest, $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$. In order to accomplish this purpose, backpropagation is based on 4 equations. The first one computes the error from the last layer:

$$\delta^L = \nabla_a C \odot f'\left(z^L\right) \tag{27}$$

where L is the last layer, $\nabla_a C$ is a vector whose components are the partial derivatives $\frac{\partial C}{\partial a_j^L}$ and $\odot$ is the Hadamard product.

The second equation computes δ^l in terms of δ^{l+1}, i.e., backpropagates the error:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot f'(z^l) \tag{28}$$

Putting both equations together we can obtain δ^l for every layer of the NN.

The third equation computes the derivative of the cost with respect to the biases:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{29}$$

The fourth equation computes the derivative of the cost with respect to the weights:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{30}$$

The four backpropagation equations are obtained from using the chain rule.
Putting all together, we can summarize how a NN works in 4 steps:

1. Given an input, compute:$z^l = \left(w^l a^{l-1} + b^l\right)$ and $a^l = f(z^l)$.
2. Compute the error vector from the last layer: $\delta^L = \nabla_a C \odot f'\left(z^L\right)$.
3. Backpropagate the error.
4. Compute the gradient of the cost function: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.

4.2 RECURRENT NEURAL NETWORKS

Recurrent neural networks (RNN) are a type of ANN whose architecture is adapted to process sequential data such as text or time series. The main difference between RNN and MLP is that RNN architecture is made with the purpose that at every instant of training it takes into account previous instants of time. The graphical representation of a RNN is the following:

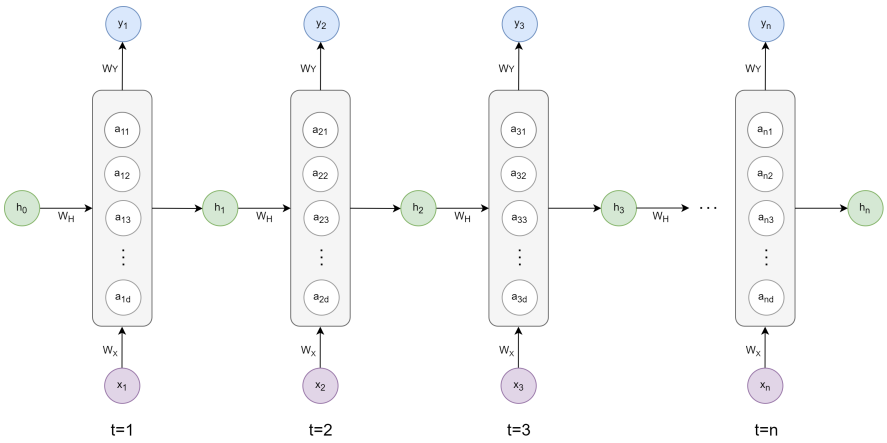


Figure 5: RNN diagram.

where a_{ti} represents the value of the i neuron at time t , x_t is the input at time t, y_t the output at time t and W_X, W_X and W_H are weight matrices constant at time. h_t is the hidden vector. h_t is responsible of transferring information from time $t-1$ to time t. It has the same size as the number of neurons and h_0 is a vector of zeros. The compact representation of a RNN is shown in Figure 6:

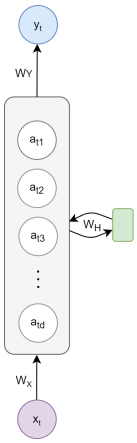


Figure 6: Diagram of a compact RNN.

Thus, at time t the NN receives two inputs: x_t and h_{t-1}, hence we can define the vector a_t as:

$$a_t = W_H h_{t-1} + W_X x_t \tag{31}$$

Likewise, at every time the NN will have two outputs: h_t and y_t:

$$h_t = tanh(a_t) \tag{32}$$

$$y_t = softmax(W_Y h_t) \tag{33}$$

Regarding to backpropagation algorithm, due to the recurrent nature of the weights, the computation of the gradients is a bit more complex. The procedure carried out is called backpropagation through time, and it follows the same mechanics (chain rule) as backpropagation in MLP. The gradient of the cost function for each weight matrix can be written as it follows (Chen 2016):

$$\frac{\partial C}{\partial W_Y} = \sum_t \frac{\partial C}{\partial y_t} \frac{\partial y_t}{\partial W_Y} \tag{34}$$

$$\frac{\partial C}{\partial W_H} = \sum_{t=1}^{n} \sum_{k=0}^{n} \frac{\partial C_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_H} \tag{35}$$

$$\frac{\partial C}{\partial W_X} = \sum_{t=1}^{n} \sum_{k=0}^{n} \frac{\partial C_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W_X} \tag{36}$$

Due to the architecture behind a simple RNN, it is very likely that the NN has a very common issue in the field of NN: The vanishing gradient problem. As the weight updates are obtained trough the partial derivative of the cost function with respect to the weight matrix, and the computation of this derivative comes from the product of a string of partial derivatives, if these elements have as result numbers smaller than 1, the final result will be a very small number, thus the weights will barely be updated. This issue gets worse through time because the factor $\frac{\partial h_{t+1}}{\partial h_k}$ from the previous equations will be the result of a product of more and more partial derivatives.

$$\frac{\partial h_{t+1}}{\partial h_k} = \frac{\partial h_{t+1}}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \cdots \frac{\partial h_{k+1}}{\partial h_k} \tag{37}$$

For this reason we say that simple RNNs have poor long-term memory.

However, there are currently complex recurrent network structures designed to address this problem, and therefore capable of having long-term memory. This is the case of Long Short Term Memory (LSTM) and Gated Recurrent Units (GRU).

4.2.1 Long Short Term Memory

LSTM introduces the concept of memory state, i.e., a vector that moves trough time like hidden state, responsible of giving long-term memory to the network. The diagram of a LSTM is shown in Figure 7.

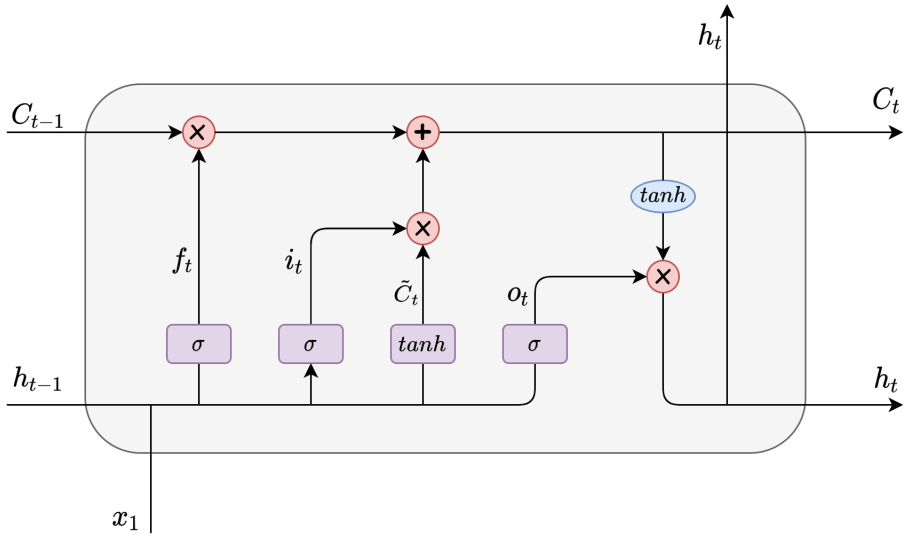


Figure 7: LSTM diagram.

As it can be seen, input and hidden state are passed through 4 neuron layes: f_t, i_t, $\tilde{C}_{t-1}$ and o_t. f_t, known as forget gate, is essential to understand the role of the memory state:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{38}$$

As forget gate is the result of a sigmoid (values between 0 and 1), if it has values near zero, the memory state from previous time will be discarded, and if it has values near 1, the memory state will be important at present. Thus, the less important past information is discarded and the more important is saved. $\tilde{C}_{t-1}$(candidate gate) and i_t(input gate) are responsible of deciding which will be the new information added to the memory state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{39}$$

$$\tilde{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{40}$$

The purpose of i_t is to find the values that will be updated, and the purpose of $\tilde{C}_t$ is to find new candidates to include to the memory state. As result, the output of the memory state will be:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t \tag{41}$$

The output gate (o_t) function is to decide witch parts of the memory state are included in the output:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \tag{42}$$

The resulting hidden state will be:

$$h_t = o_t \cdot tanh(C_t) \tag{43}$$

4.2.2 Gated Recurrent Units

The GRU architecture is simpler than the LSTM, thus it is also faster. This is because in this architecture we have 3 internal layers, and we go without memory state. The GRU diagram is as follows:

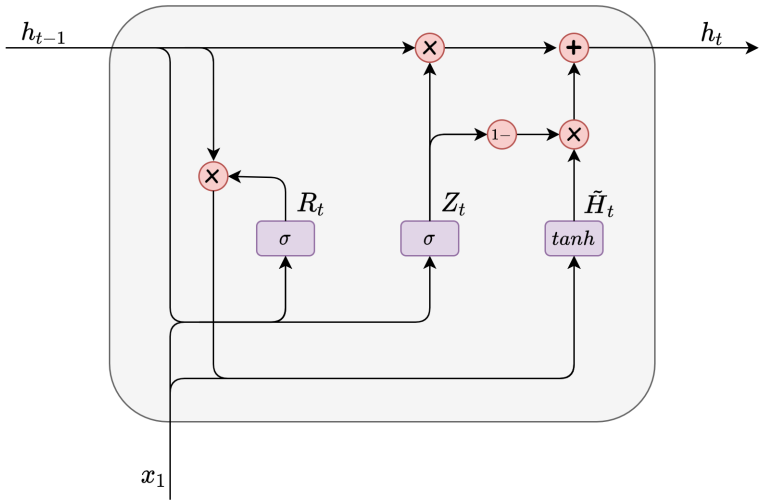


Figure 8: GRU diagram.

The three layers are: reset gate, update gate and candidate hidden state. The purpose of the reset gate is to control which amount of the previous hidden state we want to remember.

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r) \tag{44}$$

The purpose of the update gate is to find which parts of the new state are a copy of the previous:

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z) \tag{45}$$

The candidate hidden state follows a mechanic similar to the ones in LSTM:

$$\tilde{H}_t = tanh(X_t W_{xh} + (R_t \odot H_{t-1}) W_{hh} + b_h) \tag{46}$$

Lastly, to compute the hidden state, we have:

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \tilde{H}_t \tag{47}$$

4.3 CONVOLUTIONAL NEURAL NETWORKS

Convolutional Neural Networks (CNN) is a DL algorithm whose architecture is built with the aim of recognizing common patterns in two-dimensional data. Its main application is image classification, having done a great progress in numerous areas such as self

driving cars or tumor detection. In order to apply the algorithm, first we write the input as two-dimensional, as it is commonly done with this approach.

$$X = (x_{ij})_{ij=1,\dots,n} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1,(n-1)} & a_{1n} \\ x_{21} & x_{22} & \cdots & x_{2,(n-1)} & a_{2n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{(n-1),1} & x_{(n-1),2} & \cdots & x_{(n-1),(n-1)} & x_{(n-1),n} \\ x_{n1} & x_{n2} & \cdots & x_{n,(n-1)} & x_{nn} \end{pmatrix} \tag{48}$$

Input data is connected to the convolutional layer via local receptive fields. These are input areas connected with a neuron of the convolutional layer, via its corresponding weights and biases:

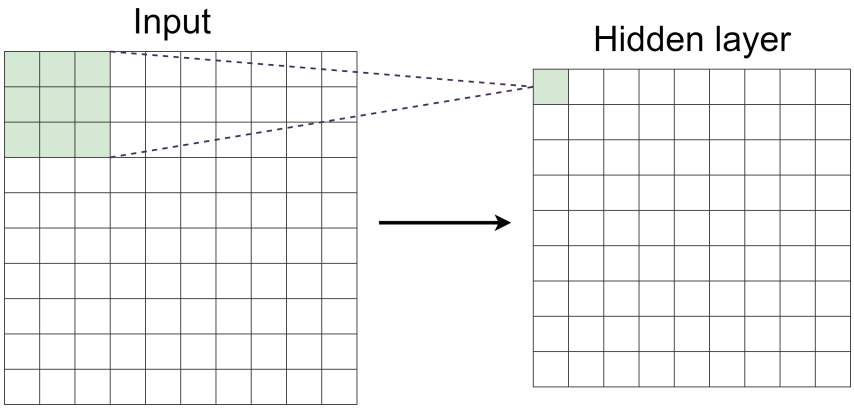


Figure 9: Diagram of a 3x3 receptive field.

The local receptive field will slide throughout the input until create the hidden layer, called feature map. On a regular basis the sliding is from neuron to neuron. Thus, if input has a $i \times j$ size and local receptive field has a $k_1 \times k_2$ size, the feature map size will be $\big(i - (k_1 - 1)\big) \times \big(j - (k_2 - 1)\big)$.

Moreover, the matrix of weights and biases, called kernel, will be the same for each of the neurons from the hidden layer. This characteristic makes possible to detect patterns in space, since with this approach each neuron of the feature map will be in charge of detecting the same characteristic. In mathematical terms, the feature map is obtained as follows: Being H_{ij} the ij neuron of the feature map, we have:

$$H_{ij} = f\left(b + \sum_{m=0}^{k_1-1} \sum_{n=0}^{k_2-1} K_{m,n} X_{i+m,j+n}\right) \tag{49}$$

where X is the input matrix and K the kernel of $k_1 \times k_2$ size.

Since a single feature map will learn a single concrete pattern, convolutional layers are composed of a large number of feature maps in order to capture all patterns useful for making predictions.

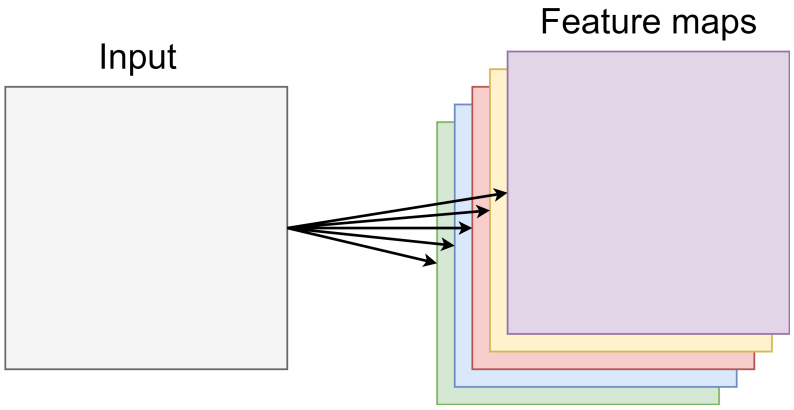


Figure 10: Diagram of a convolutional layer with many feature maps.

Furthermore, a pooling layer is usually incorporated after the convolutional layers to simplify its output (Zhang et al. 2020, chap. 6.5). For this purpose, the pooling layer summarizes the information of a certain area of the feature map. One of the most commonly used methods for this purpose is max pooling, where for a given area of the feature map the output will be the highest activation of that area.

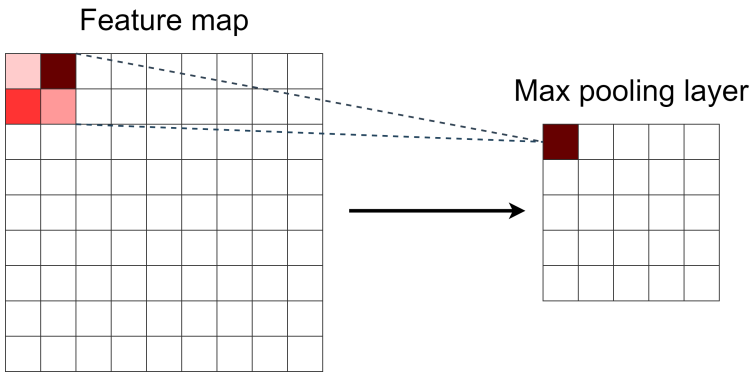


Figure 11: Max pooling diagram.

Additionally, the CNN usually ends with a flat layer of neurons. In this way, the convolutional part of the NN is specialized in feature extraction, and the flat layer in classification. The complete diagram of a CNN is the following:

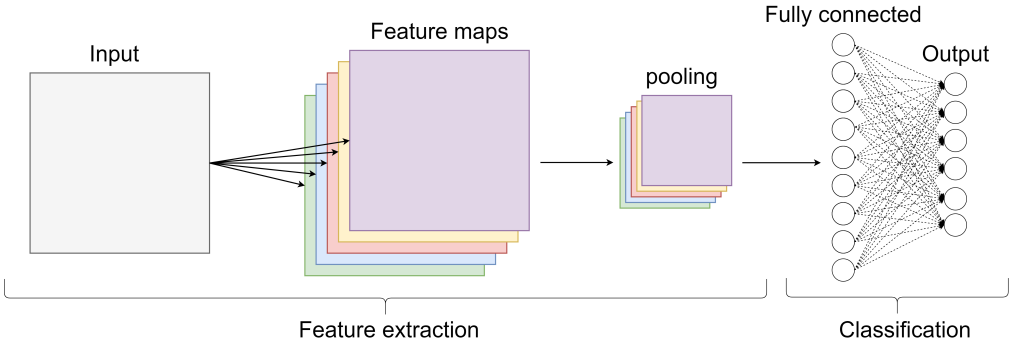


Figure 12: Diagram of a CNN.

5 DATA ANALYSIS

5.1 DATABASE

In order to feed the models and making predictions we will need the corresponding time series to electricity price and the potential exogenous variables useful to explain price. Data has been obtained from *e.sios*, a web page from Red Eléctrica de España where we can obtain data from 2014 to the present time.

As the purpose of the models is to make predictions for each day of 2020 and data has daily updates (publishing the 24 hours at a time), our models have to predict every time next 24 hours in a row. However, due to daylight savings time, not every day is 24 hours long, as last Sunday in March has 23 hours and last Sunday in October has 25. In order to make our models predict always the 24 hours of the next day, we modify our time series in such a way that every day has 24 hours: for last Sunday in March, data at 2:00 am will be the same as at 1:00 am (as first lag has highest correlation), and for last Sunday in October we remove second data at 2:00 am. It is worth noting that since we have 8760 data in a non-leap year, this change will not have a significant effect on the predictions.

5.2 EXOGENOUS VARIABLE SELECTION

For this work, a series of exogenous variables that are likely to improve predictions have been chosen. The main exogenous variables used in scientific literature are: wind energy production, total demand, total energy production, solar energy production, exports and imports (Galdeano & Basterra 2017, Cruz et al. 2011). All of them are accessible through the *e.sios* website. We can see the different correlations between them via a correlation matrix:

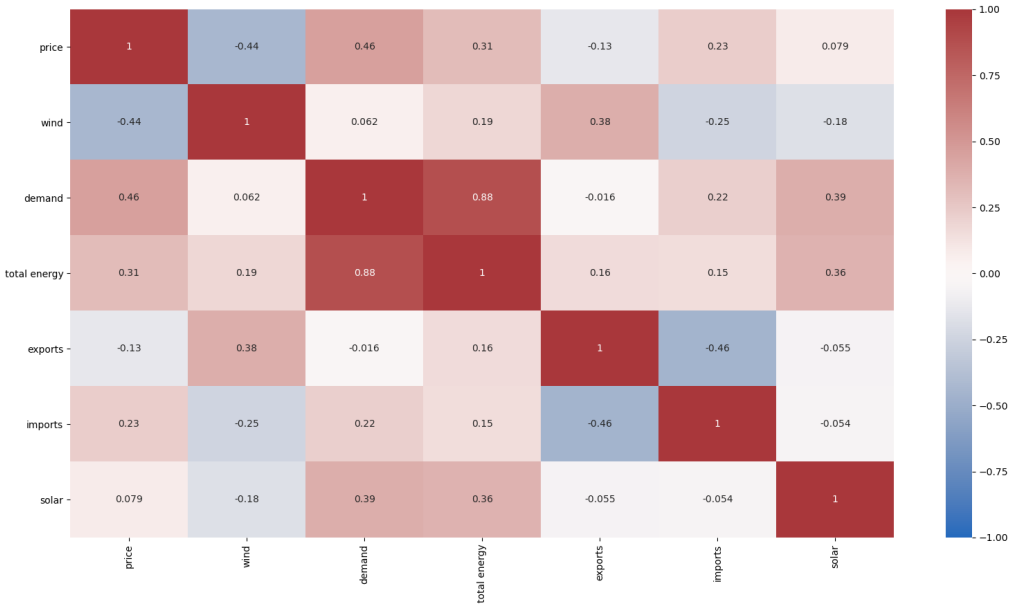


Figure 13: Correlation matrix of the exogenous variables. Code in Appendix B.1.

As it can be seen, most correlated variables are wind energy production, total demand and total energy production. These results correspond to those found in the literature.

With respect to wind energy production, as indicated by Cruz et al. (2011): “The wind generation is a fundamental driver of Spanish electricity spot prices due to its high level of penetration and its special market regulation. The inclusion of this variable dramatically improved the predictive accuracy of the dynamic regression model”. On the other hand, for total demand and total energy production, highly correlated variables, Cruz et al. (2011) write: “The electricity load forecast allows us to account for calendar effects which in other way should be captured through dummy variables or any other special treatment.”

5.3 DESCRIPTIVE ANALYSIS

In this section we perform a descriptive analysis in order to understand the behavior of the price series, and thus take into account essential components when building the models as the different types of seasonality.

5.3.1 Hourly behavior

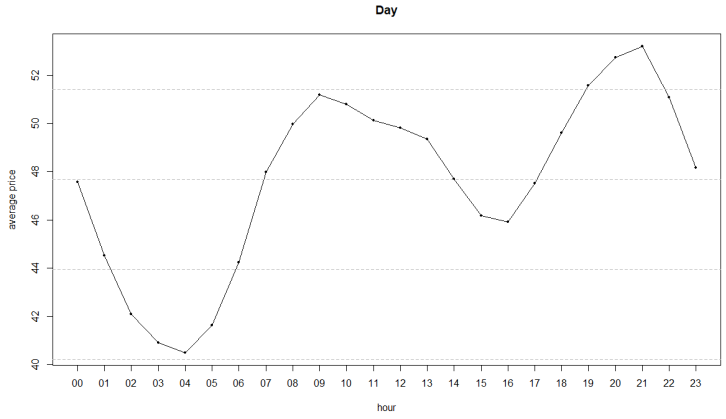


Figure 14: Average price for hours of the day. Code in Appendix B.2

During the day we can see a solid pattern: the price is lower at the evening hours, increasing as the workday begins until reaching a peak at 9:00 am. Then, price decreases slightly in the early afternoon and then increases until reaching a peak at 9:00 pm. As we can see, this pattern is closely related to the working day and daily activity schedule and therefore, to the electricity demand.

5.3.2 Daily behavior

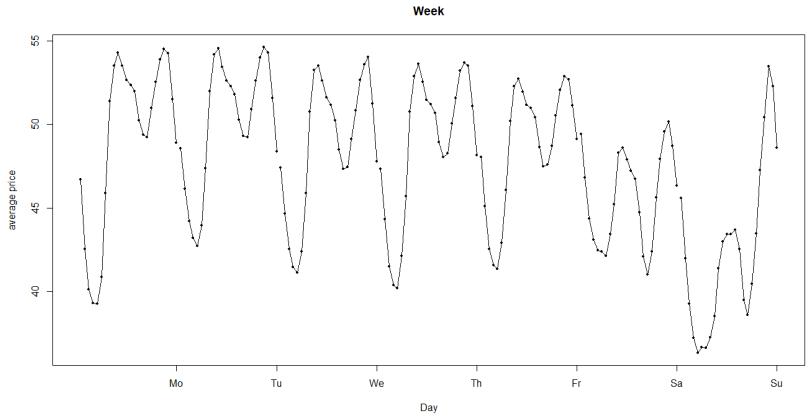


Figure 15: Average price for hours of the week. Code in Appendix B.2

The most noticeable pattern here is the difference between typical working days (Monday to Friday), that have similar patterns, and weekends, where price drops. Again, we have another evidence that supports the demand dependency on the price.

5.3.3 Seasonal behavior

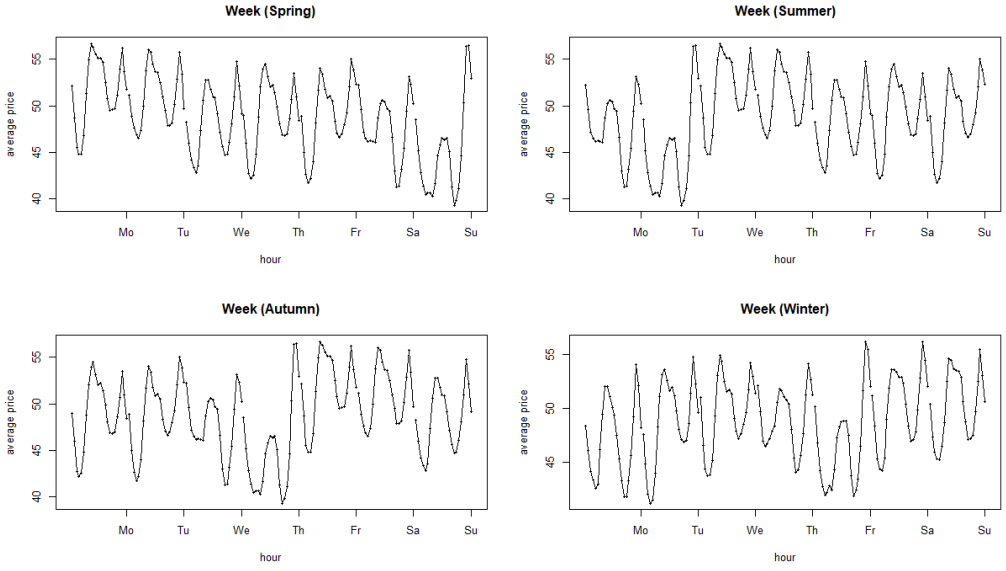


Figure 16: Average price for hours of the seasons. Code in Appendix B.2

As it can be seen, weekly behavior changes depending on the season. This behavior may suggest that studying differences between seasons such as weather and holidays may be helpful in order to build the models.

5.4 OUTLIER ANALYSIS

A noticeable characteristic in electricity price time series is the large number of outliers. In this section we suggest an Encoder-Decoder model for anomaly classification. The

Encoder-Decoder model has been successfully used for anomaly classification in different tasks (see Braei & Wagner 2020), even outperforming state of the art models in time series problems (Fengming et al. 2017). The idea behind the Encoder-Decoder is the following: the model is made up of an encoder, that aims to learn the basic characteristics of the series, and a decoder, that aims to project those characteristics into a reconstructed series.

We will label as an anomaly data whose difference between its value from the real series (x_i) and its value from the reconstructed series ($\hat{x}_i$) is larger than a threshold (δ), i.e., $e_i > \delta$ where $e_i = |x_i - \hat{x}_i|$. We use GRU layers as they are faster to train (see Fengming et al. 2017). The following diagram explains the architecture of the Encoder-Decoder:

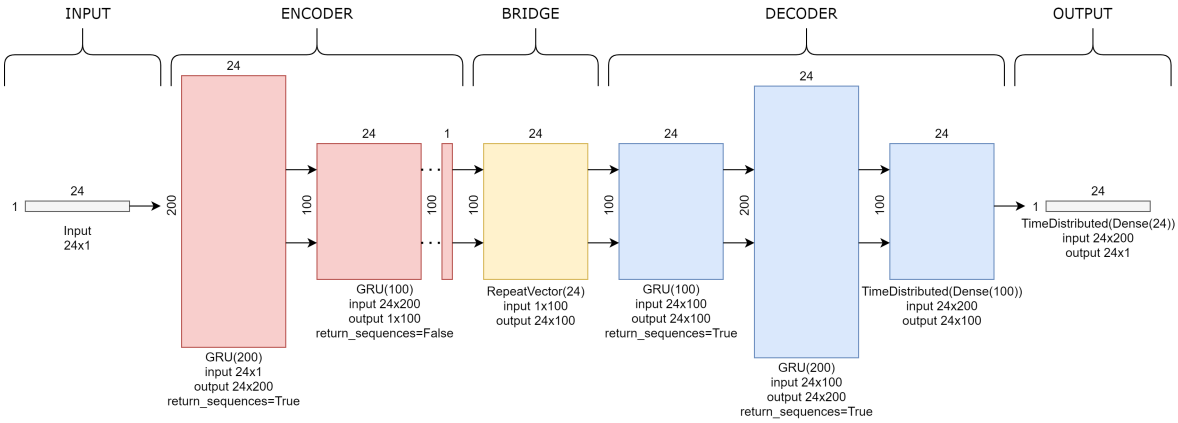


Figure 17: Diagram of an Encoder-Decoder.

Code for this section can be found in Appendix B.3. `Keras` library allows an intuitive implementation of the Encoder-Decoder due to the `RepeatVector` and `TimeDistributed` layers as well as the `return_sequences` argument.

With the Encoder-Decoder we detect outliers depending on how well the model reconstructs the data. Once the train set is reconstructed, we compute the error e and select the threshold. If a value of the error for the test set is larger than the threshold, it can be interpreted as if there was a pattern which is not familiar for the model, thus it will be labeled as an anomaly.

We train our model with data from 2018-2019 and make predictions for 2020 in order to explain outliers in test set once the predictive models are built. For this reason, it is worth mentioning that the purpose of this section is purely descriptive, and thus the information obtained will not be used for predictive purposes.

5.4.1 Explanation of some outliers

In this section we study some outliers in order to understand why are they produced. To do so, we will make use of supply and demand graphs. Code for the graphs in this section can be found in Appendix B.4.

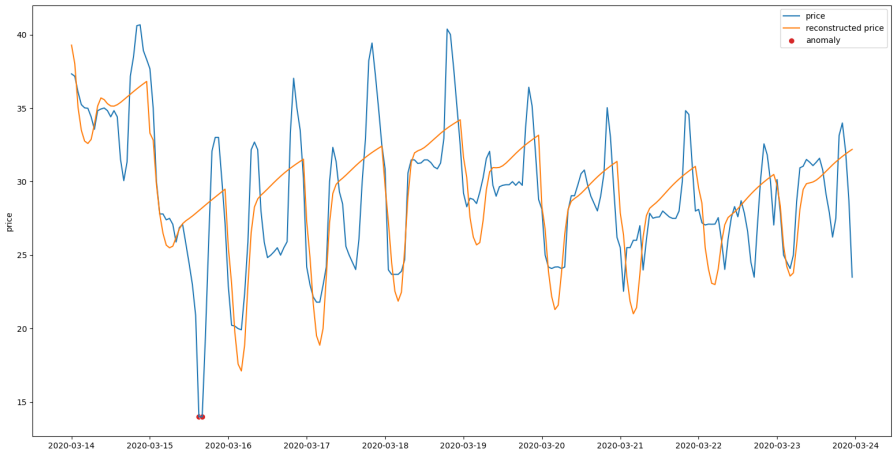


Figure 18: Outliers coinciding with the beginning of Spanish Covid-19 lockdown.

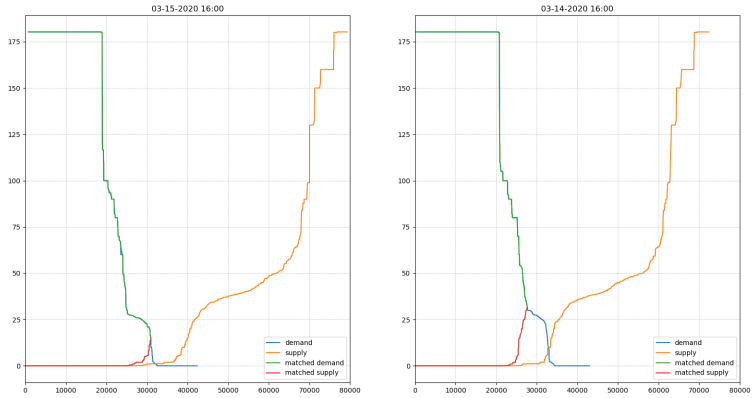


Figure 19: Comparison with price 24 hours ago.

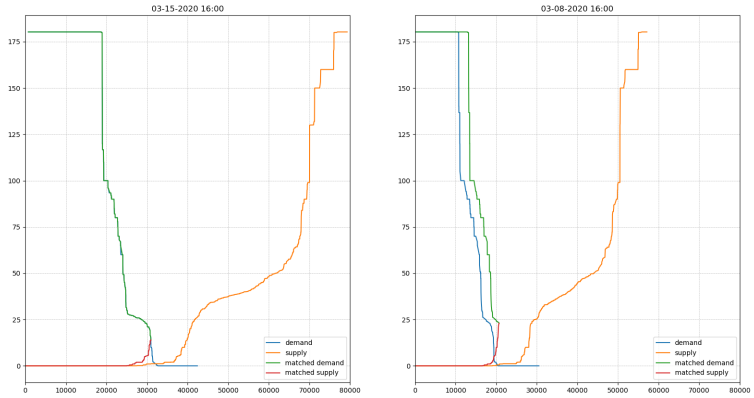


Figure 20: Comparison with price 168 hours ago.

This anomaly corresponds with some hours on March 15, coinciding with the start of Spanish lockdown due to Covid-19 pandemic.

- Supply: shifts to the right. This may be the result of wind energy supply volatility: at 16:00, wind energy supply was 11.262 MWh on March 15, 3.139,8 MWh on March 14, and 8.159,9 MWh on March 8.
- Demand: Compared to the previous day, demand remains stable. Compared to the previous week, demand shifts considerably to the right. This may be explained by the general strike on March 8.

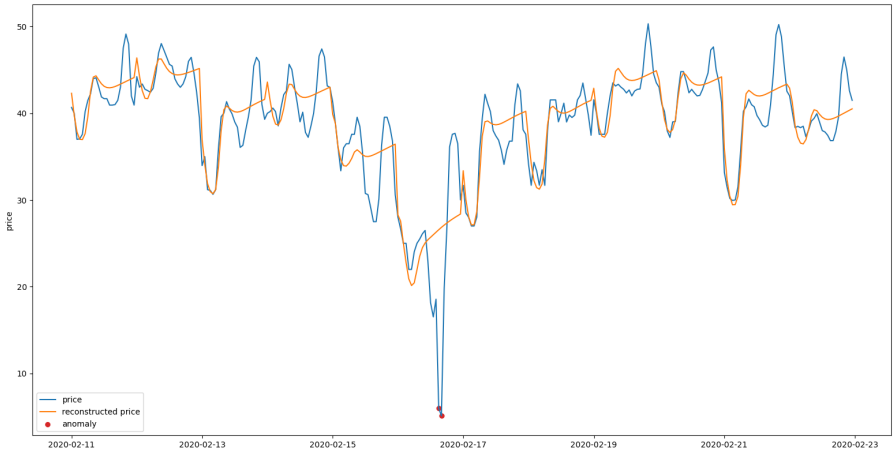


Figure 21: A day in February.

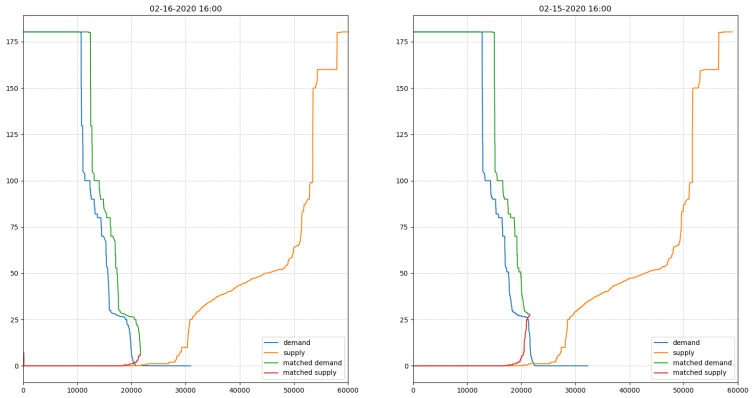


Figure 22: Comparison with price 24 hours ago.

- Supply: Slight shift to the right. Wind energy supply at 16:00 is 10859MWh at February 16 and 7431MWh at February 15.
- Demand: Slight shift to the left, explained maybe because of weekend effect, as we are comparing a Saturday with a Sunday.

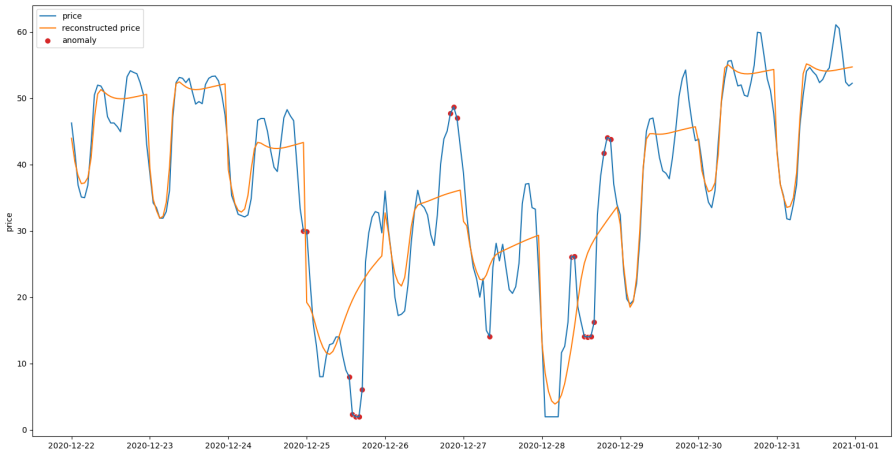


Figure 23: Christmas days.

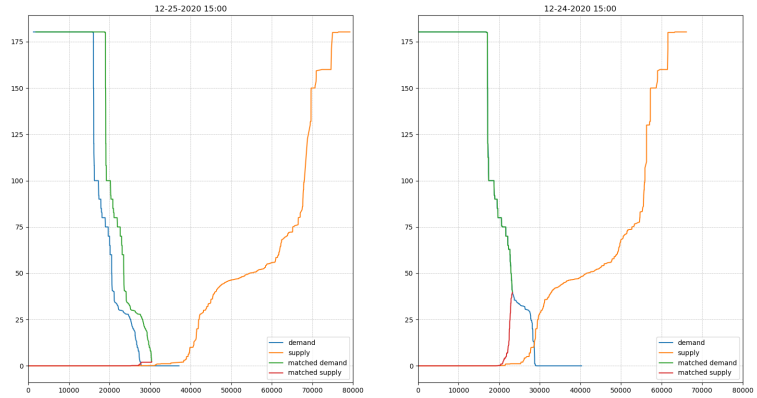


Figure 24: Comparison with price 24 hours ago.

- Supply: Large shift to the right. Wind energy supply at 15:00 is 17445MWh at December 25 and 726MWh at December 24.

- Demand: Slight shift to the left.

As we can see, when an anomaly is detected, it is likely to have changes in both demand and supply. As we know, demand is more stable than supply, so changes in demand are usually less visible. On the other side, supply is highly dependent on wind energy production, as its production has high volatility and it supplies energy at a low price.

6 ARIMA MODELS PROPOSED FOR FORECASTING

In this section we explain the process behind building the ARIMA models. Code for this section can be found in Appendix C. As in Box and Jenkins methodology, we split the procedure in three stages: Identification, estimation and diagnostic checking.

6.1 IDENTIFICATION

As defined by Box et al. (2015), "by identification we mean the use of the data, and of any information on how the series was generated, to suggest a subclass of parsimonious models worthy to be entertained". Thus, in this section we aim to pick some potential values for p, d and q and therefore create a tentative model. In order to do so, our main tools will be the Autocorrelation and Partial Autocorrelation Functions.

6.1.1 Stationarity

Before identifying the ARMA structure, we need to make the series stationary. In order to reduce heterocedasticity we can apply a logarithm to the series.

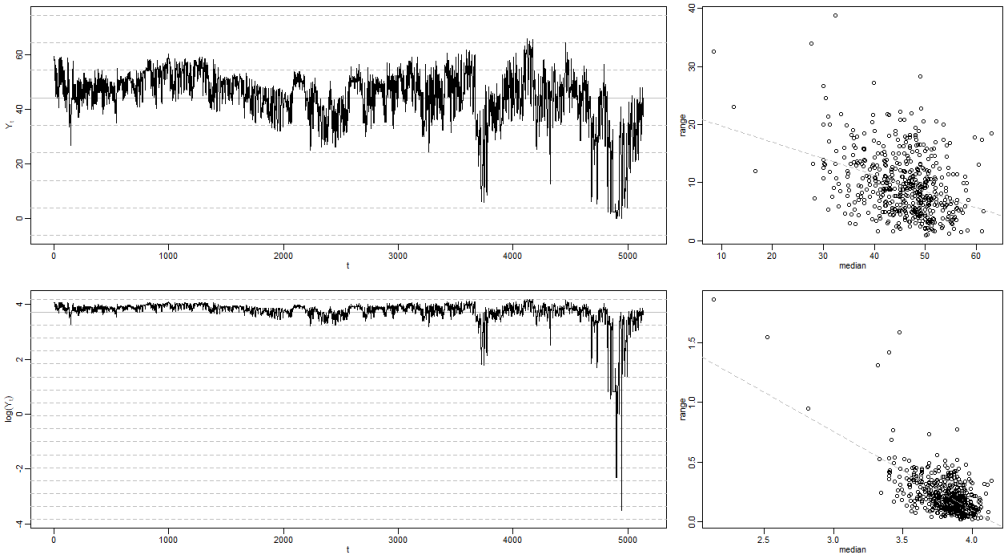


Figure 25: Series graphs with and without logarithm and its corresponding range-median graphs.

As it can be seen in the range-median graphs, applying a logarithm doesn't seem to ameliorate heterocedasticity. To be sure about that, we can implement the Bartlett's test which aims to verify homoscedasticity. Bartlett's test hypotheses and Bartlett's test statistic are the following:

$$H_0 : \sigma_1^2 = \sigma_2^2 = \cdots = \sigma_k^2; \quad H_1 : \sigma_i^2 \neq \sigma_j^2 \tag{50}$$

$$H = \frac{(n-k)\log(s^2) - \sum_{j=1}^{k}(n_j - 1)\log(s_j^2)}{1 + (1/(3(k-1)))((\sum_{j=1}^{k} 1/(n_i - 1)) - 1/(n-k))} \tag{51}$$

Where we have k samples with sizes n_j, s_j^2 as the variance of the jth group, n as the sum of all samples sizes ($n = \sum_{i=1}^{k} n_j$), and s^2 as the pooled variance, $s^2 = \sum_{i=1}^{k}(n_j - 1)s_j^2/(n-k)$.

Both p-values for the series and the logarithm of the series are statistically zero therefore applying logarithms does not solve heterocedasticity problem. However, even if it does not solve the heterocedasticity problem, applying logarithms can improve model results. In order to verify this, we have done predictions for 2019 with the models with and without logarithms and conclude that the model without logarithms is better overall.

After discarding applying a logarithm, next step is to have a look at the AFC and PACF in order to figure out if the series has trend and seasonality:

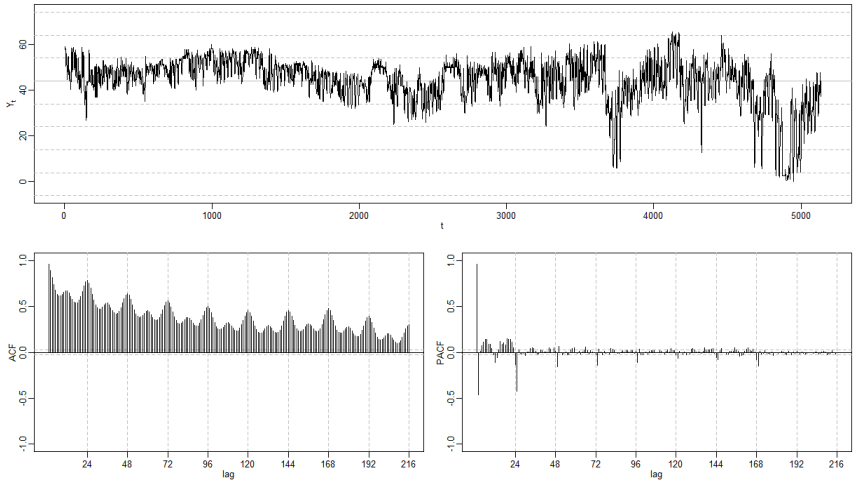


Figure 26: ACF and PACF for the series.

By looking at the AFC we can see clearly a pattern corresponding to daily seasonality. As we know from the data analysis section, this series should have both daily and weekly seasonalities. Since we also see a trend, we apply a regular difference in order to see seasonality more clearly:

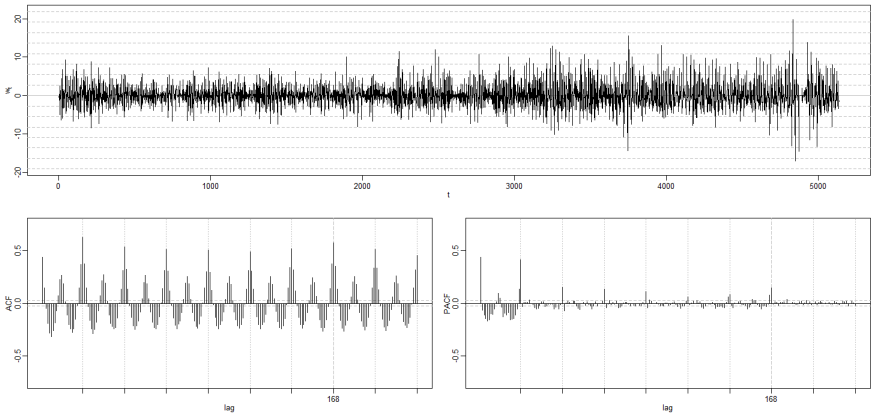


Figure 27: Series and its ACF and PACF after applying differences in lag 24 .

As it can be seen it looks that both daily and weekly seasonalities are required.

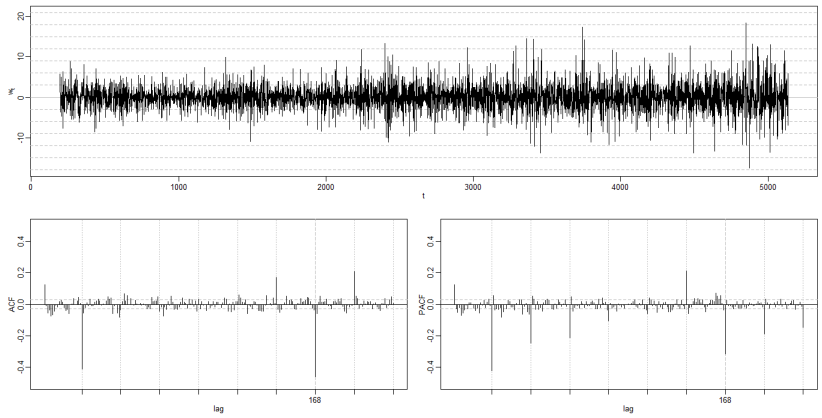


Figure 28: Series and its ACF and PACF after applying differences in lags 1, 24 and 168.

As it can be seen, despite heteroscedasticity we can now consider the series as stationary and therefore continue with identifying the ARMA structure.

6.1.2 Finding ARMA structure

As we can see in Figure 28, ACF and PACF have a pattern corresponding to a MA$(1)_{24}(1)_{168}$:

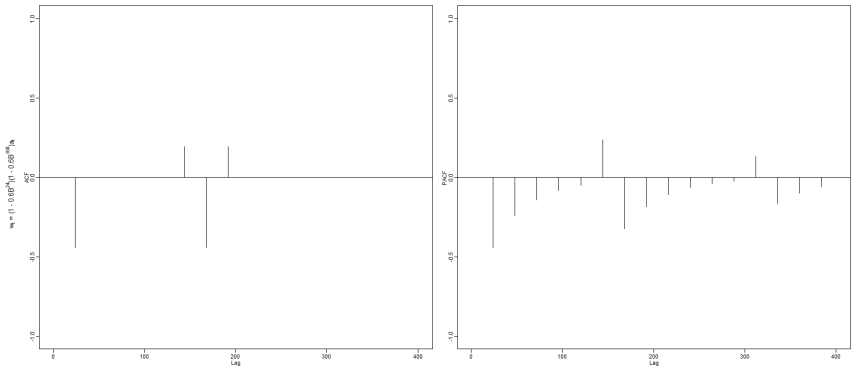


Figure 29: Theoretical ACF and PACF of a MA$(1)_{24}(1)_{168}$ process.

As in Figure 28 we can see a considerable correlation in lag 1, and in order to be sure about applying daily and weekly differences, the AR component may be AR$(1)(1)_{24}(1)_{168}$. Thus, in this section we conclude that the most tentative model is:

$$(1-\phi B)(1-\phi B^{24})(1-\phi B^{168})\nabla Y_t = (1-\theta B^{24})(1-\Theta B^{168})u_t \tag{52}$$

6.2 ESTIMATION AND DIAGNOSTIC CHECKING

In order to reduce computational cost of estimation we will use only last 8 weeks of 2019. The estimation of (52) is the following:

$$(1-0.15B)(1-0.91B^{24})(1-0.99B^{168})\nabla Y_t = (1-0.66B^{24})(1-0.81B^{168})u_t \tag{53}$$

As it can be seen, we confirm that a weekly difference is needed and daily difference seems probable. For the diagnostic checking, we may have a look at some tools given by the `diagchk` function from `tfarima` library:

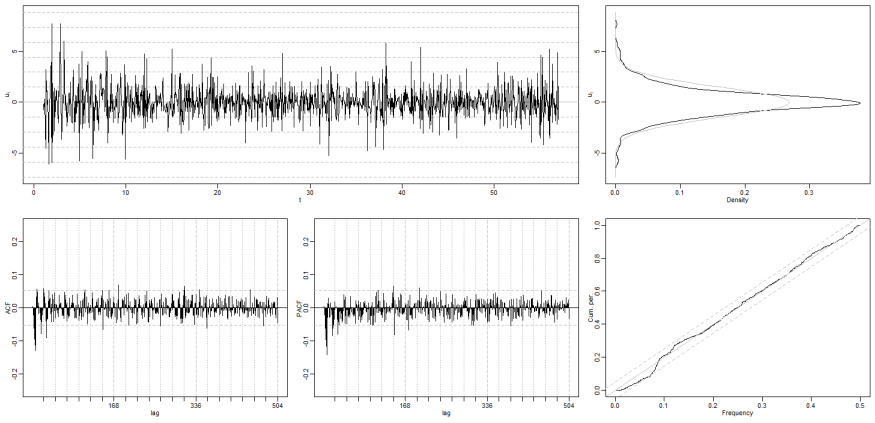


Figure 30: Diagnostic tools for the residuals of the model.

As we can see in the periodogram and in the ACF and PACF corresponding to the residuals of the model, we have some correlation problems between lags 1 to 24, thus we have to deal with it if we want to make the residuals white noise.

As we saw in data analysis section, during the day there are hourly hidden periodicities, so that may be the cause of 1 to 24 autocorrelations. In order to describe this behavior we can use periodic functions as sine and cosine. Since in 24 time-steps can be a maximum of 12 periods, we can try to capture the different hourly hidden periodicities using the cosine function in twelve AR(2) via the expression: $1 - 2\cos(2\pi f/24)\sqrt{\phi}B + \phi B^2$ where $f = 1, 2, \ldots, 12$. By doing so, the model turns to be:

$$(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)(1-\phi B^{24})(1-\phi B^{168})\nabla Y_t = (1-\theta B^{24})(1-\Theta B^{168})u_t \tag{54}$$

And its corresponding graphs for diagnostic checking are:

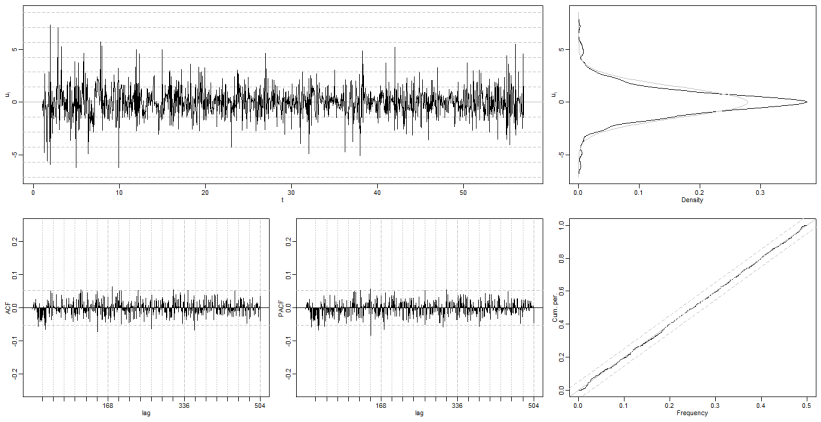


Figure 31: Diagnostic tools for the model.

As we can see, autocorrelation issue seems to be solved. The model still has heterocedaticity and the residuals aren't exactly distributed as a normal distribution, so we will have to carry with this problems. In spite of this, this model is yet appropriate for predict.

Fitting model (54) to all the data we estimate the following model:

$$(1-0.85B)(1-1.8B+0.89B^2)(1-1.7B+0.92B^2)(1-1.3B+0.88B^2)(1-0.93B+0.87B^2)$$
$$(1-0.48B+0.86B^2)(1-1.1\cdot 10^{-16}B+0.86B^2)(1+0.48B+0.86B^2)$$
$$(1+0.91B+0.83B^2)(1+1.3B+0.83B^2)(1+1.6B+0.82B^2)(1+1.8B+0.84B^2)$$
$$(1+0.91B)(1-0.98B^{24})(1-B^{168})\nabla Y_t = (1-0.9B^{24})(1-0.92B^{168})u_t \quad (55)$$

As the AR$(1)_{168}$ and AR$(1)_{24}$ parameters estimations are close to 1, we may use both daily and weekly differences. Thus, the final model expression can be written as:

$$(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)\nabla\nabla^{24}\nabla^{168}Y_t = (1-\theta B^{24})(1-\Theta B^{168})u_t \quad (56)$$

Diagnostic checking graphs for the final model are the following:

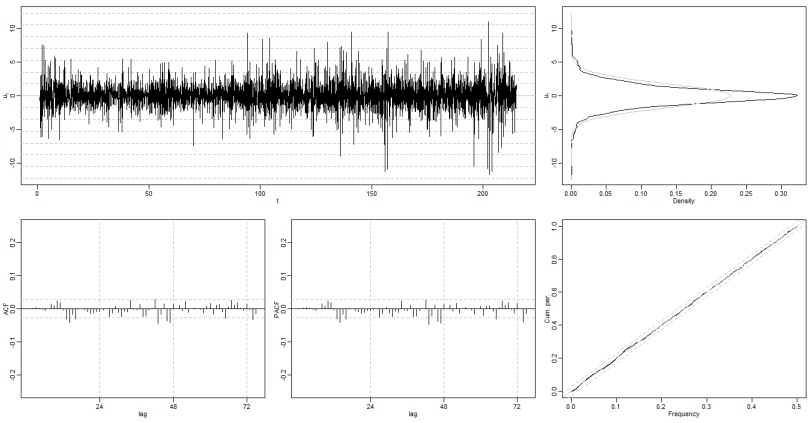


Figure 32: Diagnostic tools for the model.

As the same as in Figure 31, we still have heterocedaticity and high kurtosis.

6.3 TRANSFER FUNCTION MODELING

In this section we propose a Transfer Function Model with the aim of making predictions with a multivariate input. In order to find the parameters for a Transfer Function Model, we will use the Cross-Correlation function of the residuals of the univariate model and the residuals of the prewhitening input model. By prewhitening input model we mean a model for the input whose residuals are white noise. We will use demand (X_1) and wind (X_2) as inputs, thus both series have been prewhitened. The corresponding model for demand is:

$$(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)\nabla\nabla^{24}\nabla^{168}X_{1t} = (1-\theta B^{24})(1-\Theta B^{168})u_t \quad (57)$$

And for wind we have:

$$(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)\nabla\nabla^{24}X_{2t} = (1-\theta B^{24})u_t \quad (58)$$

Once we have the prewhitened models, we can use its residuals to compute the Cross-correlation functions. For demand we have:

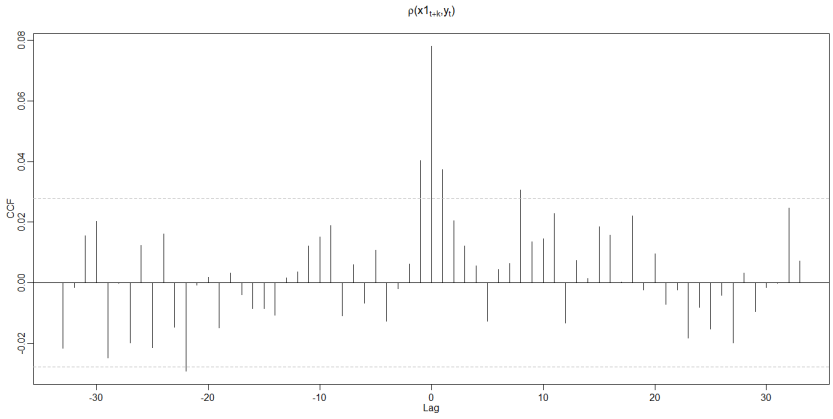


Figure 33: Cross-correlation function for demand and price.

As it can be seen, most significant lag is on 0. For wind we have:

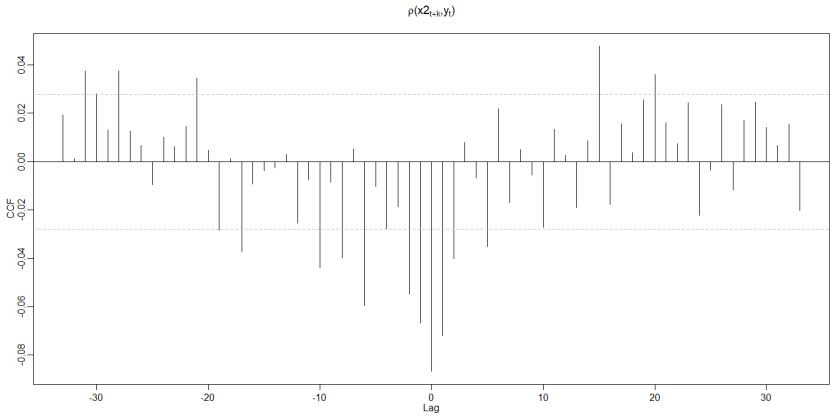


Figure 34: Cross-correlation function for wind and price.

As it can be seen, it seems that we have an AR process starting at lag 0.

Since the most significant correlations are in the first lags and in practice we do not have access to demand and wind data for the forecast period, we will have to make predictions of both series for 24 steps ahead and use the values from these predictions. With all this, we can write the Transfer Function Model as follows:

$$Y_t = \omega_{0_1}\nabla\nabla^{24}\nabla^{168}X_{1t} + \frac{\omega_{0_2}}{(1-\delta B)}\nabla\nabla^{24}X_{2t} + \frac{(1-\theta B^{24})(1-\Theta B^{168})}{(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)}u_t \tag{59}$$

Graphs corresponding to diagnostic checking are:

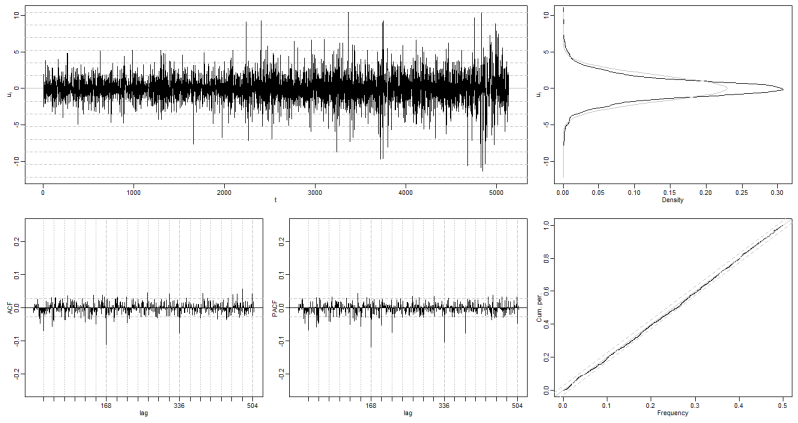


Figure 35: Diagnostic tools for the model.

As it can be seen, there are significant correlations in lags 168 and 336. To minimize them, we can incorporate a MA$(2)_{168}$. Therefore, we can write the final model as:

$$Y_t = \omega_{0_1}\nabla\nabla^{24}\nabla^{168}X_{1t} + \frac{\omega_{0_2}}{(1-\delta B)}\nabla\nabla^{24}X_{2t} + \frac{(1-\Theta B^{24})(1-\Theta B^{168})(1-\Theta B^{168}-\Theta B^{336})}{(1-\phi B)\prod_{f=1}^{12}(1-2\cos(2\pi f/24)\sqrt{\phi}B+\phi B^2)}u_t \tag{60}$$

Graphs corresponding to diagnostic checking are:

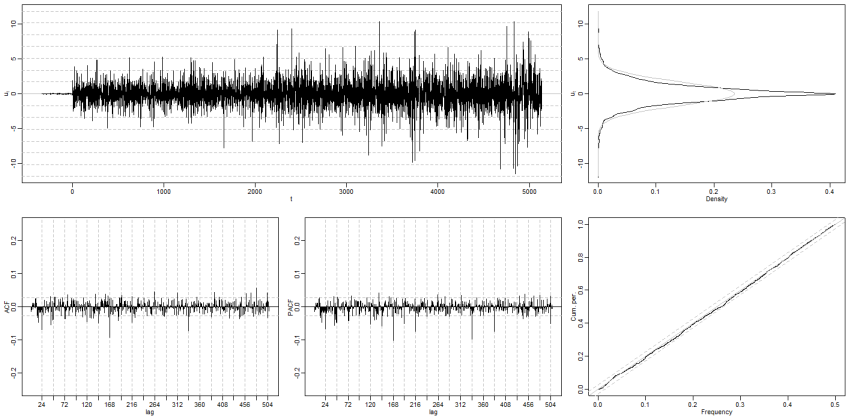


Figure 36: Diagnostic tools for the model.

As we can see, there are still significant autocorrelations thus the residuals are not white noise. In addition, we have the problems of high kurtosis and heterocedasticity as in the univariate model.

7 NEURAL NETWORKS MODELS PROPOSED FOR FORECASTING

7.1 ENVIRONMENT FOR SUPERVISED LEARNING

In order to apply NN for time series prediction, we need to reorder the data with a supervised learning structure. For that purpose we use the sliding window method (Brownlee

2018, chap. 4). With this approach, inputs will be the chosen lags to predict, and outputs will be the corresponding predictions. Moreover, it is necessary to split the dataset in train set and test set. We will also use a validation set in order to tune hyperparameters. Train set will be data from 2014 to 2018. 2019 will be validation set and 2020 test set. The following libraries and versions have been used for the development of the code: `Numpy`(1.19.2), `Pandas` (1.1.3), `Scikit-learn` (0.23.2), `Matplotlib` (3.3.2), `Seaborn` (0.11.1), `Tensorflow` (2.0.0), `Keras` (2.3.1) and `Keras-tuner` (1.0.1).

7.2 NEURAL NETWORKS OPTIMIZATION

Hyperparameter optimization as well as the implementation of different approaches and tools are essential in order to build a competitive NN. In this section we discuss the different procedures carried out with the aim of improving its performance. All the procedures have been performed using the validation set.

7.2.1 Choosing the strategy for multi-step ahead forecasting

In order to choose the best strategy for multi-step ahead forecasting two strategies have been evaluated (see Taieb et al. 2010): As in ARIMA, we can use the Recursive Multi-step Forecast, where we use a one-step model multiple times where the prediction for the prior time step is used as an input for making a prediction on the following time step, or predict 24 timesteps at a time (Multiple Output Strategy), therefore going without the t_{-1} lag.

Both strategies have been developed (see Appendix D.1), and it has been shown that Multiple Output Strategy generates better predictions. This result is consistent with findings in (Taieb et al. 2012), where different multi-step ahead forecasting strategies are performed for the 111 time series of the NN5 forecasting competition, concluding as follows : “The most consistent findings are that Multiple-Output approaches are invariably better than Single-Output approaches.” Consequently, the final model will be developed with a multi-step ahead strategy.

7.2.2 Lags selection

In order the choose the right lags, we can start from the knowledge acquired when specifying the ARIMA model. As we know, once t_{-1} lag has been discarded, two most significant lags are the ones that corresponds with daily seasonality (t_{-24}) and weekly seasonality (t_{-168}). Furthermore, the following lags have been tested: 48, 72, 96, 120, 144.

By testing the different combinations, it has been determined that the lags used in the final model will be lags 24, 168 and 144, since with this combination better predictions are obtained for the validation set.

7.2.3 Exogenous variables

For the model with exogenous variables, the addition of variables suggested in data analysis section to the model have been tested, using data at day before (24 time steps). It has been concluded that the features that most improve the predictions for the validation set are wind and demand.

7.2.4 Data normalization

Data normalization is a widely used tool that improves predictions and reduces training time (Shanker et al. 1996). We will rescale values into a range of [-1,1].

7.2.5 Neurons activation function

We will use ReLu (Rectified Linear Unit) activation function, defined as:

$$f(z) = max(0, z) \tag{61}$$

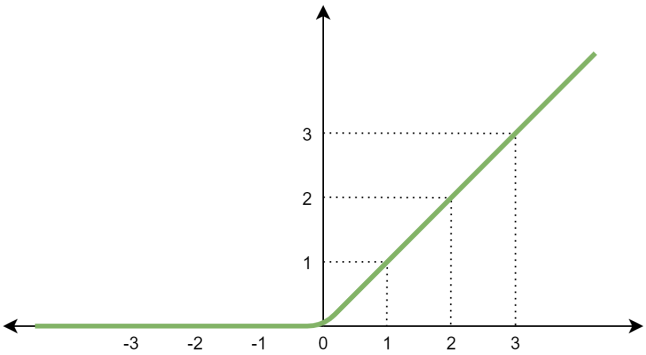


Figure 37: Graphical representation of a ReLu.

This activation function has been proven to provide faster training due to a non-saturation of its gradient (Krizhevsky et al. 2012).

7.2.6 Learning rate schedule

Learning rate is one of the most important hyperparameters. Setting an optimal value is essential for developing a competitive NN, since as it is explained by Goodfellow et al. (2016, chap. 11.4): “When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. […] When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error.”

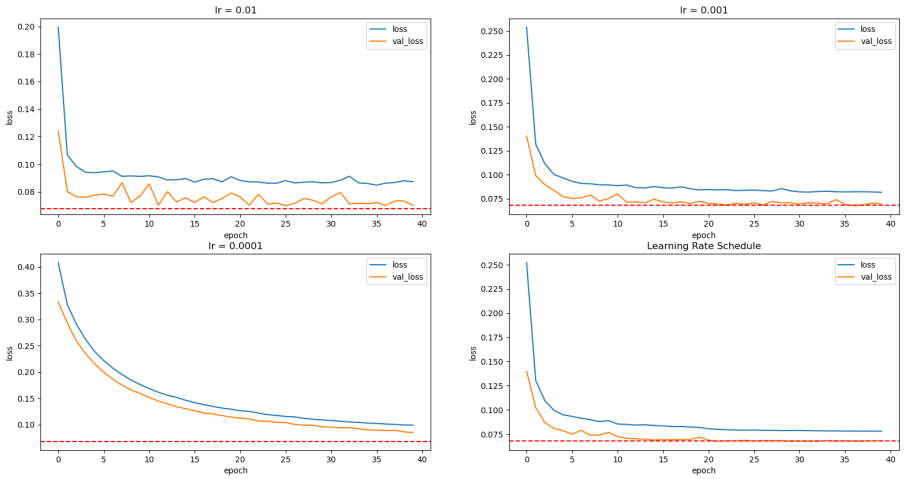


Figure 38: Diferent values for learning rate.

To solve this issue it is common to use a learning rate schedule: “In practice, it is necessary to gradually decrease the learning rate over time ” (Goodfellow et al. 2016, chap. 8.3.1). We have developed a learning rate schedule for each of the NN.

7.2.7 L2 regularization

We use L2 regularization, as it can prevent overfitting and give more consistency to the algorithm. Implementing L2 regularization, the cost function can be written as follows:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{62}$$

Where C_0 is the original cost function, n the size of the training set , and λ the regularization parameter. As overfitting is not a main issue in our time series, we will not give a high value to this hyperparameter because we could jeopardize predictions.

7.2.8 Retraining

Due to the daily publication of electricity prices, we can retrain the models with more data once it is published. This procedure is essential when a small amount of data is available, but for our time series, where we have 52584 data for the 2014-2019 period, retraining the model with 24 more data will not result in a noticeable improvement, and it will have a high computational cost. It has been decided to retrain the model on a quarterly basis.

7.2.9 Ensembling

In Neural Networks, variance, defined as the amount that the estimate of the target function will change given different training data, is usually high. Due to the random initialization of weights, every time we train a NN it will fall into a different local minima. If we train the NN several times and save its corresponding predictions, we can then join all of them, for example by averaging, creating what we call a committee. The predictions of the committee are usually better than the best single network used in isolation (Bishop et al. 1995, chap. 9). We have used a committee for each of the NN proposed using model averaging ensemble.

7.2.10 Hyperparameters optimization

Hyperparameter optimization is one of the biggest challenges in order to build a competitive NN. At this moment, there are some heuristics when searching for good hyperparameters (see Nielsen 2015, chap. 3) that are indispensable. However, manual tunning also requires spending a lot of hours in front of the computer.

To automate this process, we use `keras tuner`, a powerful tool from `keras` created specifically for this purpose. In Appendix D.2 code for tunning some hyperparameters of the RNN is developed.

7.3 NEURAL NETWORKS ARCHITECTURES PROPOSED

The following sections explain the different particularities of the NN architectures tested. Code for this section can be found in Appendix D.3 and explanatory diagrams of each architecture can be found in Appendix E. Since in `Keras` the input is three dimensional, [samples, time steps, features], it is worth noting that the only difference between the univariate model and the multivariate model is the incorporation of two more features (wind and demand at day before).

7.3.1 Convolutional Neural Network

It is worth noting that as we use a 1D convolutional layer, the kernel does not have to be squared as the best-known (used in classification tasks) explained in Neural Networks models section, since the dimensions of it is: number of rows $\times$ kernel size. This design is specific to adapt convolutional layers to time series analysis. We also apply padding, as data from the corners is as important as data in the middle. The ensembling used has 20 individual CNN for both univariate and multivariate models.

7.3.2 Recurrent Neural Network

We use a GRU instead of LSTM as it is faster to train and also obtains better results in the validation set. We apply L2 regularization with a parameter smaller than the one used in CNN as if we increase it predictions would be worse and overfitting does not seem a big issue for validation set. The ensembling used has 10 individual RNN for both univariate and multivariate models.

7.3.3 Hybrid CNN-RNN model

We have combined CNN and RNN in order to make predictions. The main idea of this approach is that CNN can be used as the encoder in an encoder-decoder architecture (Brownlee 2018, chap. 20.9). The ensembling used has 10 individual CNN-RNN for both univariate and multivariate models.

8 NEURAL NETWORKS AND ARIMA ENSEMBLING

In this section a combination between ARIMA and NN is proposed with the aim of creating and ensembled model that outperforms each individual model. This idea is supported by one of the most fundamental findings that Makridakis et al. (2020) obtained in the M5 competition: “The M5 Accuracy competition confirmed the findings of the previous four M competitions and those of numerous other studies suggesting that combining forecasts of different methods, even relatively simple ones, results in improved accuracy”.

Regarding model specification, we have tested all the architectures for NN in the validation set and concluded that CNN outperforms the rest. Testing ARIMA for 2019, we found that CNN also outperforms it, however, it is worth highlighting that NN hyperparameters are biased to this year (as it was used for hyperparameter optimization), thus the difference between CNN and ARIMA might not be as significant for the test set. Hence, we will use an equal-weighted combination for both ARIMA and CNN models. This approach, although very simple, is commonly used among researchers: As an example, in the M5, the top 3 ranked teams used equal-weighted combination for their different ensemblings (Makridakis et al. 2020). Code for this section can be found in Appendix F.

9 RESULTS

In this section we will discuss the predictive power of the developed models. In order to measure the error, we will use three of the most used metrics:

- Root mean square error:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{p}_i - p_i)^2}$$

- Mean absolute error:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$$

- Mean Absolute Percentage Error:

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{|y_i - \hat{y}_i|}{|y_i|}100$$

9.1 UNIVARIATE MODELS RESULTS

In this section we will discuss the results of the proposed univariate models. We have created a naive model (see Appendix G.1) in order to provide a benchmark. The main results for the different metrics are the followings:

	RMSE	MAE	MAPE
NAIVE	7.462	5.364	23.083
ARIMA	5.116	3.756	16.059
CNN	5.034	3.714	17.009
RNN	5.218	3.942	17.251
CNN-RNN	5.118	3.802	17.295

Figure 39: Table with proposed models and its error metrics. Code in Appendix G.2

In first place, it is worth noting the better performance of CNN compared to the rest of the NN architectures. This is an unexpected result since, in general, RNN, whose architecture is focused on the prediction of sequential data, have better performance in prediction problems with time series.

In second place, it is worth highlighting the good performance of the CNN compared to the ARIMA, obtaining better results in 2 of the 3 error metrics. Since the global result of the error metrics is similar between these two models, a further study will be realized in the next section.

Before that, we will try to understand why CNNs performs so well. In order to do so, we have represented the weights for the kernels as follows:

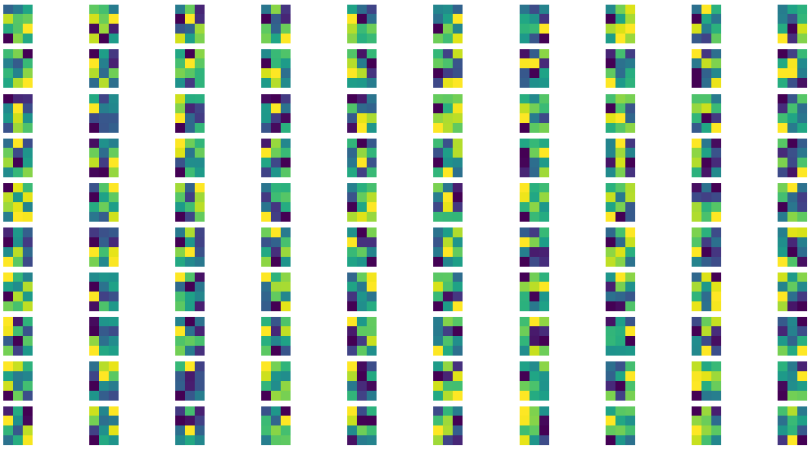


Figure 40: CNN kernels for one of the CNNs. Code in Appendix G.3

As it can be seen, the allocation of weights does not seem to follow an easily recognizable pattern. This problem, i.e., explaining causality, is for most researchers the worst problem in DL, and has led to the popularization of the ”black box” concept when referring to the learning process behind DL algorithms (see Castelvecchi 2016).

However, in order to explain the good performance of CNNs, a hypothesis for future research is proposed: Due to the series multiple seasonality, it is likely that a pattern that occurred a week ago (weekly seasonality), the day before (daily seasonality), or a few hours ago (intra-daily seasonality) will happen again in the future. The architecture of CNN is designed for pattern recognition in space, hence it seems reasonable to think that it performs well at identifying the different seasonalities, which, by expressing the input as a matrix as we have done, can be construed as patterns in space.

9.1.1 Detailed analysis of the results

As we do 24 step ahead predictions, each of them has a different prediction horizon, thus we will differentiate the error according to the different hours in a day. For the following results, we have chosen RMSE as the error metric. First, we compare the NN models to evaluate the degree of dominance that CNNs has over the rest:

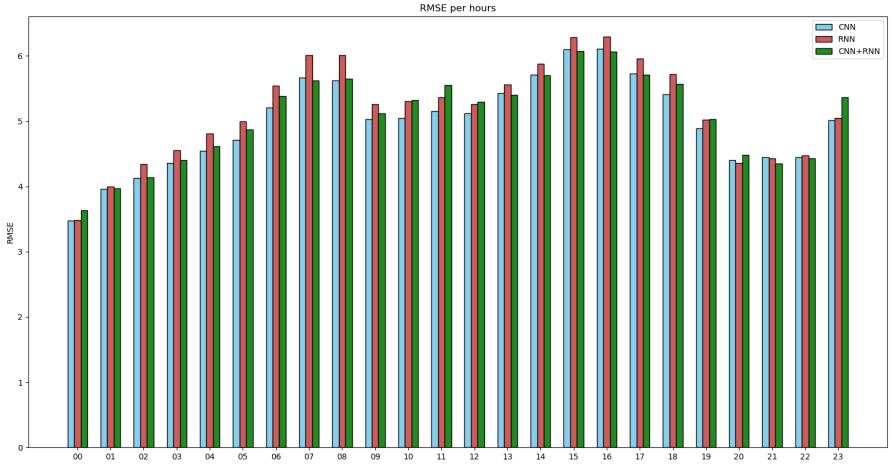


Figure 41: RMSE per hour for the NN models. Code in Appendix G.2.

As we can see, CNNs seems to perform better overall, thus we select this NN to make the comparison with the ARIMA model.

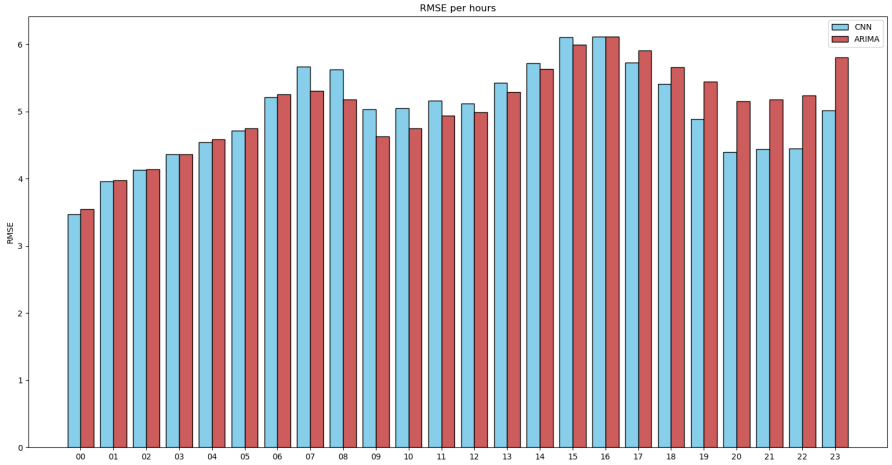


Figure 42: RMSE per hour for ARIMA and CNN. Code in Appendix G.2.

As we can see, the ARIMA outperforms the CNNs in the hour range from 07:00 to 15:00, corresponding to the first price spike throughout the day seen in Data Analysis section. However, for the 18:00 to 23:00 hour range, corresponding to the second price spike throughout the day, CNNs perform significantly better.

In addition to the RMSE per hour, RMSE per day and month has also been computed:

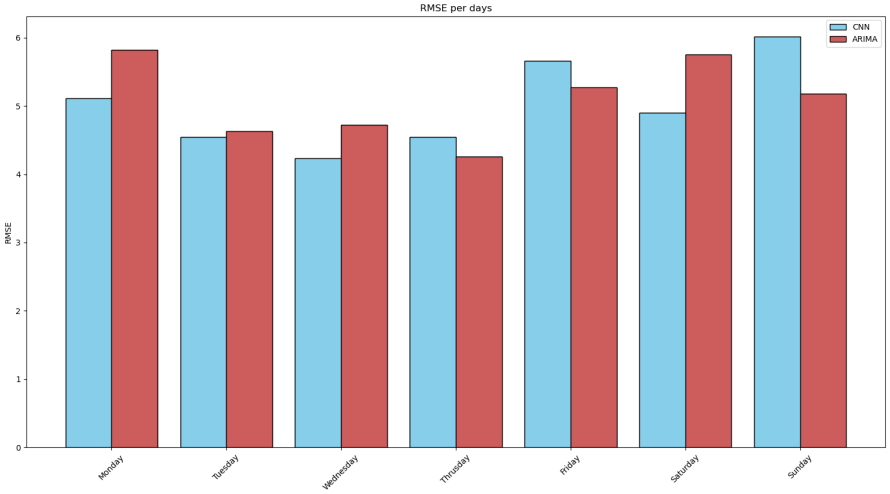


Figure 43: RMSE per days for ARIMA and CNN. Code in Appendix G.2.

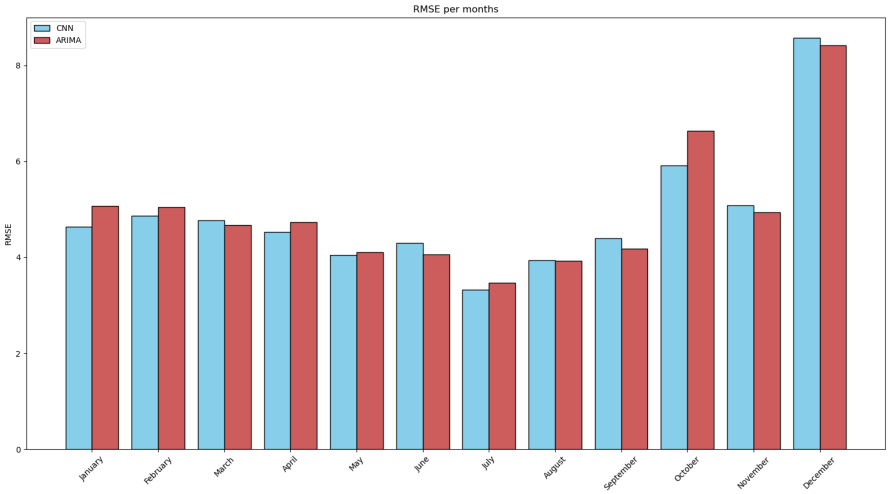


Figure 44: RMSE per months for ARIMA and CNN. Code in Appendix G.2.

Regarding the RMSE per days, it is worth noting the significant differences on Saturday, Sunday and Monday. For the RMSE per months, the most significant difference is in October, which is the month with the most outliers detected by the Encoder-Decoder.

9.2 MULTIVARIATE MODELS RESULTS

With respect to the multivariate analysis, on one hand it is worth noting that the results for the Transfer Function model have not been able to outperform those of the ARIMA. In addition to this fact, it is worth mentioning that the construction of the Transfer Function Model has specification complexities (namely making the exogenous series stationary and finding the values for s, r and b) while the aggregation of the exogenous variables to the NN model only involves a change in the number of features that feeds the input.

On the other hand, it is worth noting that CNN is the only NN model that significantly improves over the univariate results. To see the dominance of it with respect to the other

NN models, we can represent the following graph:

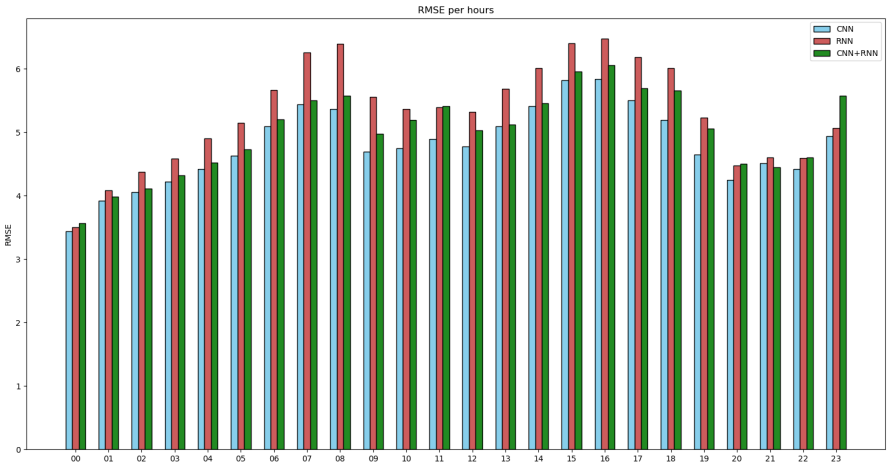


Figure 45: RMSE per hour for the NN multivariate models. Code in Appendix G.2

As we can see, the inclusion of exogenous variables increases the gap between CNN and the other NN models, suggesting that this model is better for capturing the properties of exogenous variables. The comparison of the RMSE per hours between the multivariate CNN and the ARIMA is as follows:

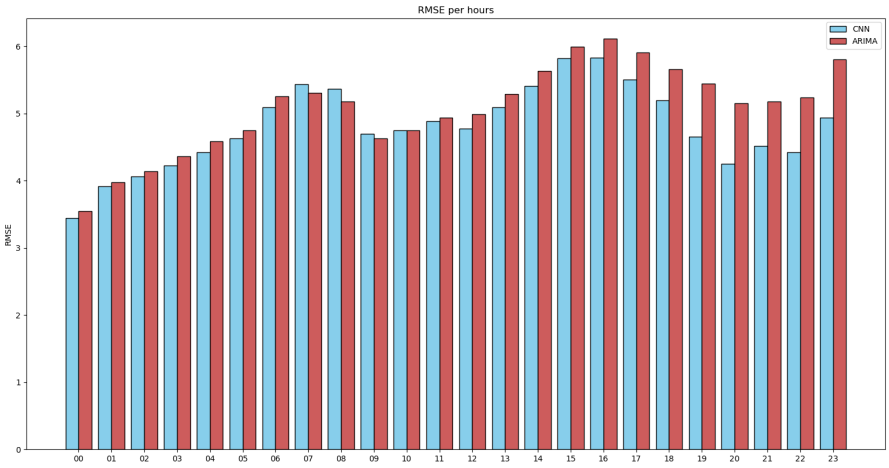


Figure 46: RMSE per hour for ARIMA and multivariate CNN. Code in Appendix G.2

As we can see, the multivariate CNN model outperforms the ARIMA overall.

9.3 ARIMA AND CNN ENSEMBLING

If we compare the ensembling model between ARIMA and CNN with each of its components, i.e., the ARIMA model and the multivariate CNN model, the results for the different metrics are as follows:

	RMSE	MAE	MAPE
ARIMA-CNN	4.646	3.426	15.107
CNN	4.841	3.602	16.170
ARIMA	5.116	3.756	16.059

Figure 47: Table with error metrics. Code in Appendix G.2

As we can see, the ARIMA-CNN model considerably outperforms the individual models. Analyzing the RMSE for the different hours per day we have:

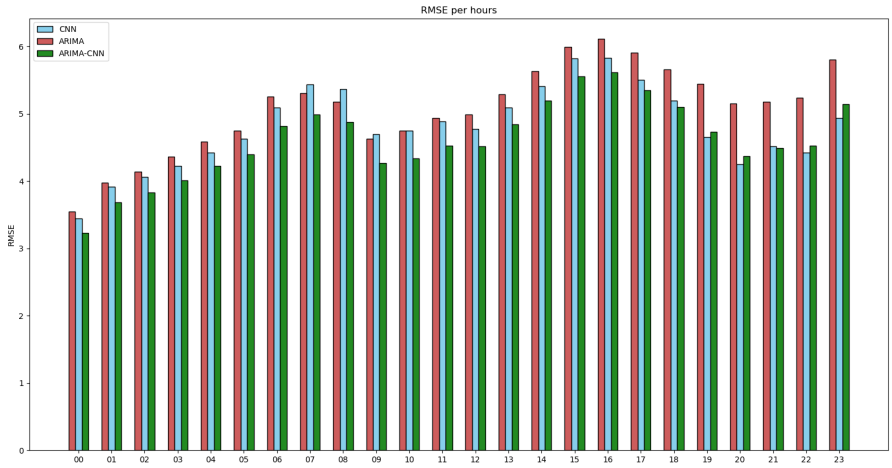


Figure 48: RMSE per hour for ARIMA, multivariate CNN and ARIMA-CNN. Code in Appendix G.2

Overall, we can say that the combination of both models generates better predictions than the individual ones. To get and idea of the importance of having a background in these two approaches for time series prediction, we provide a simple quantitative approximation of the benefits it can bring: Hong (2015) makes an estimation of the dollar amount saved by a typical medium-size utility with a 5-GW peak load, concluding that a 1% improvement in price forecasting accuracy (roughly the amount that ensembling model improves over individual models), would lead to a savings of approximately $1.5 million per year.

10 CONCLUSION

In this work, both ARIMA and NN models have been developed with the aim of evaluating their predictive power for electricity price in the Spanish electricity market at univariate and multivariate levels, as well as suggesting an ensembling model that gathers the best models of both approaches. The main findings of the work are the following:

In the first place, it is worth mentioning the good performance of the CNN model proposed. With respect to the other NN models, it outperforms them at both univariate and multivariate levels. With respect to the ARIMA models, it has similar results at univariate level and outperforms it at multivariate level. It is also worth highlighting that CNN is the model which improves the most when introducing wind and demand variables. As future work, it is suggested to find out what the reasons behind the good performance of CNNs are and testing them for more high-seasonal time series in order to give consistency to the hypothesis proposed in section 9.1.

In the second place, it has been proven that an ensembling of both ARIMA and NN models significantly outperforms the results of each individual model, supporting the findings of Makridakis competitions.

In the third place, several differences between both approaches have been shown:

- Hyperparameter optimization and ensembling make the specification of the NN model more time-consuming.
- The addition of exogenous variables in NN is straightforward, while in ARIMA it has more complexities.
- ARIMA explains causality better than NN, where causality is one of the main problems.
- Lags selection in NN is based on ARIMA methodology.

All these findings suggest that a background on Deep Learning would be beneficial for time series prediction, as it has been proven to be a powerful tool with still a lot of room for improvement.

References

Bishop, C. M. et al. (1995), *Neural networks for pattern recognition*, Oxford university press.

Box, G. E. & Jenkins, G. M. (1976), *Time series analysis: forecasting and control*, Holden Day.

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

Braei, M. & Wagner, S. (2020), 'Anomaly detection in univariate time-series: A survey on the state-of-the-art', *arXiv preprint arXiv:2004.00433* .

Brownlee, J. (2018), *Deep learning for time series forecasting: predict the future with MLPs, CNNs and LSTMs in Python*, Machine Learning Mastery.

Castelvecchi, D. (2016), 'Can we open the black box of ai?', *Nature News* **538**(7623), 20.

Chen, G. (2016), 'A gentle tutorial of recurrent neural network with error backpropagation', *arXiv preprint arXiv:1610.02583* .

Cruz, A., Muñoz, A., Zamora, J. L. & Espínola, R. (2011), 'The effect of wind generation and weekday on spanish electricity spot price forecasting', *Electric Power Systems Research* **81**(10), 1924–1935.

Fengming, Z., Shufang, L., Zhimin, G., Bo, W., Shiming, T. & Mingming, P. (2017), 'Anomaly detection in smart grid based on encoder-decoder framework with recurrent neural network', *The Journal of China Universities of Posts and Telecommunications* **24**(6), 67–73.

Galdeano, U. C. & Basterra, M. L. (2017), 'Determinantes del precio de la electricidad en españa', *Estadística española* **59**(194), 119–149.

Goodfellow, I., Bengio, Y., Courville, A. & Bengio, Y. (2016), *Deep learning*, Vol. 1, MIT press Cambridge.

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., Liu, T., Wang, X., Wang, G., Cai, J. et al. (2018), 'Recent advances in convolutional neural networks', *Pattern Recognition* **77**, 354–377.

Hong, T. (2015), 'Crystal ball lessons in predictive analytics', *EnergyBiz Mag* **12**(2), 35–37.

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012), 'Imagenet classification with deep convolutional neural networks', *Advances in neural information processing systems* **25**, 1097–1105.

Makridakis, S., Andersen, A., Carbone, R., Fildes, R., Hibon, M., Lewandowski, R., Newton, J., Parzen, E. & Winkler, R. (1982), 'The accuracy of extrapolation (time series) methods: Results of a forecasting competition', *Journal of forecasting* **1**(2), 111–153.

Makridakis, S., Spiliotis, E. & Assimakopoulos, V. (2018), 'The m4 competition: Results, findings, conclusion and way forward', *International Journal of Forecasting* **34**(4), 802–808.

Makridakis, S., Spiliotis, E. & Assimakopoulos, V. (2020), 'The m5 accuracy competition: Results, findings and conclusions', *Int J Forecast* .

Nielsen, M. A. (2015), *Neural networks and deep learning*, Vol. 25, Determination press San Francisco, CA.

Olah, C. (2015), 'Understanding LSTM Networks', https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

OMIE (2020), 'Funcionamiento del mercado diario', https://www.omie.es/sites/default/files/inline-files/mercado_diario.pdf.

Rosenblatt, F. (1958), 'The perceptron: a probabilistic model for information storage and organization in the brain.', *Psychological review* **65**(6), 386.

Shanker, M., Hu, M. Y. & Hung, M. S. (1996), 'Effect of data standardization on neural network training', *Omega* **24**(4), 385–397.

Taieb, S. B., Bontempi, G., Atiya, A. F. & Sorjamaa, A. (2012), 'A review and comparison of strategies for multi-step ahead time series forecasting based on the nn5 forecasting competition', *Expert systems with applications* **39**(8), 7067–7083.

Taieb, S. B., Sorjamaa, A. & Bontempi, G. (2010), 'Multiple-output modeling for multi-step-ahead time series forecasting', *Neurocomputing* **73**(10-12), 1950–1957.

Wani, M. A., Bhat, F. A., Afzal, S. & Khan, A. I. (2020), *Advances in deep learning*, Springer.

Zareipour, H., Canizares, C. A. & Bhattacharya, K. (2009), 'Economic impact of electricity market price forecasting errors: a demand-side analysis', *IEEE Transactions on Power Systems* **25**(1), 254–262.

Zhang, A., Lipton, Z. C., Li, M. & Smola, A. J. (2020), *Dive into Deep Learning*. https://d2l.ai.

APPENDIX

A Python code for power market and pricing section

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
pd.options.mode.chained_assignment = None


df=pd.read_csv("power_market.csv", sep=",")


ofertada = df.loc[df["ofertada_casada"] == "O"]
casada = df.loc[df["ofertada_casada"] == "C"]

demanda_normal = ofertada.loc[ofertada["compra_venta"] == "C"]
oferta_normal = ofertada.loc[ofertada["compra_venta"] == "V"]

demanda_casada = casada.loc[casada["compra_venta"] == "C"]
oferta_casada = casada.loc[casada["compra_venta"] == "V"]


def acumulada(tipo):
    tipo=tipo.reset_index()
    cantidad = tipo["cantidad"]
    cant_acu = list()
    for i in range(len(cantidad)):
        if i == 0:
            cant_acu.append(cantidad[i])
        if i > 0:
            cant_acu.append(cantidad[i] + cant_acu[i - 1])

    tipo["cantidad_acumulada"] = cant_acu
    return tipo


demanda_normal = acumulada(demanda_normal)

oferta_normal = acumulada(oferta_normal)

demanda_casada = acumulada(demanda_casada)

oferta_casada = acumulada(oferta_casada)


plt.plot(demanda_normal["cantidad_acumulada"], demanda_normal["precio"])
plt.plot(oferta_normal["cantidad_acumulada"], oferta_normal["precio"])
plt.plot(demanda_casada["cantidad_acumulada"], demanda_casada["precio"])
plt.plot(oferta_casada["cantidad_acumulada"], oferta_casada["precio"])
plt.ylabel('EUR/MWh')
plt.xlabel('Power energy')
plt.legend(['demand', 'supply',"matched demand", "matched supply"])
plt.title('03-16-2021 01:00')
```

B Code for data analysis section

B.1 Python code for correlation matrix

```
import numpy as np
import pandas as pd
```

```
import seaborn as sns

df1=pd.read_csv("precios_luz_14_19.txt", sep=",", index_col=0)
df2=pd.read_csv("eolica_14_19.csv", sep=",", index_col=0)
df3=pd.read_csv("demanda_14_19.csv", sep=",", index_col=0)
df4=pd.read_csv("energia_total_14_19.csv", sep=",", index_col=0)
df5=pd.read_csv("exptotales_14_19.csv", sep=",", index_col=0)
df6=pd.read_csv("imptotales_14_19.csv", sep=",", index_col=0)
df7=pd.read_csv("solar_14_19.csv", sep=",", index_col=0)

result = pd.concat([df1,df2,df3,df4, df5, df6, df7], axis=1)

result.corr()

sns.heatmap(result.corr(), annot=True, vmin=-1, vmax=1, cmap="vlag")
```

B.2 R Code for descriptive analysis

```
library(tfarima)

df <- read.table("precios_luz_14_19.txt",
                 header = TRUE, sep = "," , col.names = c("date", "price"))

head(df)

#price for  2019
p19 <- df$price[startsWith(df$date, "2019")]

#dates for 2019
d19 <- df$date[startsWith(df$date, "2019")]

#dates for the days
days <- substr(d19, 1, 10)

hours <- substr(d19, 12, 13)

#function to know the weekday
dow <- weekdays(as.Date(days))

#days of the week
Dow <- weekdays(as.Date("2021-03-08")+0:6)

#hours of the day
h <- c(paste("0", 0:9, sep = ""), 10:23)

#matrix 365x24. prices for each hour
ph <- sapply(h, function(x) p19[hours == x])

m <- apply(ph, 2, mean)

#PLOT
plot.ts(m, xaxt = "n", ylab = "average price", xlab =  "hour", type = "o", pch
    = 16, cex = 0.5, main = "Day")
axis(side=1, at = 1:24, labels= h)
abline(h=c(mean(m), mean(m)-sd(m), mean(m)-2*sd(m), mean(m)+sd(m)), col = "gray
    ", lty = 2)


#WEEKS

#dh: prices for each hour of the week: 168 hours in a week, 52 weeks in a year
dh <- list()
for (d in Dow)
  for (h in hours[1:24])
```

```
    dh <- c(dh, list(p19[dow == d & hours == h]))

#price mean for each hour in a week
m <- sapply(dh, mean)



#PLOT
plot(m,type = "n", xaxt="n", ylab = "average price", xlab = "Day", main = "Week
    ")
for (i in 0:6) {
  indx <- (i*24+1):((i+1)*24)
  lines(indx, m[indx])
  points(indx, m[indx], pch = 16, cex = 0.5)
}
axis(side=1, at = 24*(1:7), labels= c("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su")
    )


#SEASONS

#Dates for spring days
spring <- as.character(as.Date("2019-03-20")+(0:92))

#Prices for spring days
p19sp <- p19[days %in% spring]

#Name of the spring days
dowsp <- dow[days %in% spring]

#spring hours
hourssp <- hours[days %in% spring]

dh <- list()
for (d in Dow)
  for (h in hours[1:24])
    dh <- c(dh, list(p19sp[dowsp == d & hourssp == h]))

m <- sapply(dh, mean)


par(mfrow=c(2, 2))
plot(m,type = "n", xaxt="n", ylab = "average price", xlab = "hour", main = "
    Week (Spring)")
for (i in 0:6) {
  indx <- (i*24+1):((i+1)*24)
  lines(indx, m[indx])
  points(indx, m[indx], pch = 16, cex = 0.5)
}
axis(side=1, at = 24*(1:7), labels= c("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su")
    )



#Summer
summer <- as.character(as.Date("2019-06-21")+(0:92))
p19sm <- p19[days %in% summer]
dowsm <- dow[days %in% summer]
hourssm <- hours[days %in% summer]

dh <- list()
for (d in Dow)
  for (h in hours[1:24])
```

```
    dh <- c(dh, list(p19sp[dowsm == d & hourssm == h]))
m <- sapply(dh, mean)

plot(m,type = "n", xaxt="n", ylab = "average price", xlab = "hour", main = "
    Week (Summer)")
for (i in 0:6) {
  indx <- (i*24+1):((i+1)*24)
  lines(indx, m[indx])
  points(indx, m[indx], pch = 16, cex = 0.5)
}
axis(side=1, at = 24*(1:7), labels= c("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su")
    )



#Autumn
fall <- as.character(as.Date("2019-09-22")+(0:90))
p19f <- p19[days %in% fall]
dowf <- dow[days %in% fall]
hoursf <- hours[days %in% fall]
dh <- list()
for (d in Dow)
  for (h in hours[1:24])
    dh <- c(dh, list(p19sp[dowf == d & hoursf == h]))
m <- sapply(dh, mean)
plot(m,type = "n", xaxt="n", ylab = "average price", xlab = "hour", main = "
    Week (Autumn)")
for (i in 0:6) {
  indx <- (i*24+1):((i+1)*24)
  lines(indx, m[indx])
  points(indx, m[indx], pch = 16, cex = 0.5)
}
axis(side=1, at = 24*(1:7), labels= c("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su")
    )


#Winter
winter <- as.character(as.Date("2019-01-21")+(0:57))
winter <- c(winter,  as.character(as.Date("2019-12-21")+(0:10)))
p19w <- p19[days %in% winter]
doww <- dow[days %in% winter]
hoursw <- hours[days %in% winter]
dh <- list()
for (d in Dow)
  for (h in hours[1:24])
    dh <- c(dh, list(p19sp[doww == d & hoursw == h]))
m <- sapply(dh, mean)
plot(m,type = "n", xaxt="n", ylab = "average price", xlab = "hour", main = "
    Week (Winter)")
for (i in 0:6) {
  indx <- (i*24+1):((i+1)*24)
  lines(indx, m[indx])
  points(indx, m[indx], pch = 16, cex = 0.5)
}
axis(side=1, at = 24*(1:7), labels= c("Mo", "Tu", "We", "Th", "Fr", "Sa", "Su")
    )
```

B.3 Python code for the Encoder-Decoder

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from sklearn.metrics import mean_absolute_error
```

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

#Split data in train and test
def split_dataset(data):
  train, test = data[35064:-8784], data[-8784:]
  train = np.array(np.split(train, len(train)/24))
  test = np.array(np.split(test, len(test)/24))
  return train, test

#Encoder-Decoder model
def build_model():
    n_timesteps, n_features, n_outputs = train.shape[1], train.shape[2], train.
    shape[1]
    model = keras.Sequential()
    model.add(keras.layers.GRU(200, activation='relu', input_shape=(n_timesteps
    , n_features), return_sequences=True))
    model.add(keras.layers.GRU(100, activation="relu", return_sequences=False))
    model.add(keras.layers.RepeatVector(n_timesteps))
    model.add(keras.layers.GRU(100, activation='relu', return_sequences=True))
    model.add(keras.layers.GRU(200, activation='relu', return_sequences=True))
    model.add(keras.layers.TimeDistributed(keras.layers.Dense(100, activation="
    relu")))
    model.add(keras.layers.TimeDistributed(keras.layers.Dense(n_features)))
    opt=keras.optimizers.Adam(learning_rate=0.001)
    model.compile(loss='mae', optimizer=opt)
    history=model.fit(train, train, epochs=30, batch_size=30, verbose=1,
    validation_data=(test, test), callbacks=[callback])
    return model, history

#Learning rate schedule
def scheduler(epoch, lr):
  if epoch < 10:
    return lr
  if epoch < 20:
    return 0.0005
  else:
    return 0.0001

#Early stopping
callback2 = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3,
    verbose=1)

#Callback for learning rate schedule
callback = tf.keras.callbacks.LearningRateScheduler(scheduler)

#Read the data
df=pd.read_csv("precios_14_20.txt", sep=",", index_col=0)

#Split in train and test
train, test= split_dataset(df)

#Build autoencoder model
model, history=build_model()



#reconstructed training data
train_reconstructed=model.predict(train)

#flatten the data
train_reconstructed_flat=train_reconstructed.reshape(train_reconstructed.shape
    [0]*train_reconstructed.shape[1],)
```

```
train_flat=train.reshape(train.shape[0]*train.shape[1],)

#MAE representation
train_mae= abs(train_reconstructed_flat - train_flat)
plt.hist(train_mae, bins=300)
plt.xlabel("Train MAE")
plt.ylabel("No of samples")
plt.show()




#Reconstructed test
test_reconstructed = model.predict(test)

#Flatten data
test_reconstructed_flat=test_reconstructed.reshape(test_reconstructed.shape[0]*
    test_reconstructed.shape[1],)
test_flat=test.reshape(test.shape[0]*test.shape[1],)

#MAE representation
test_mae = abs(test_reconstructed_flat - test_flat)
plt.hist(test_mae, bins=300)
plt.xlabel("test MAE")
plt.ylabel("No of samples")
plt.show()

#set the threshold
threshold = np.std(train_mae)*4

anomalies = test_mae > threshold

#Database with prices, index and anomalies
indice=df[-8784:].index
df_test_flat=pd.DataFrame(data=test_flat, columns=["price"], index=indice)
df_test_flat["anomaly"]=anomalies

#Save database
df_test_flat.to_csv("df_test_flat.csv")

#Database with reconstructed price
df_test_reconstructed_flat = pd.DataFrame(data=test_reconstructed_flat, columns
    =["reconstructed"], index=indice)

#Save database
df_test_reconstructed_flat.to_csv("df_test_reconstructed_flat.csv")
```

B.4 Python code for outliers graphs

B.4.1 Code for reconstructed time series and outliers graphs

```
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

#Load databases
df_test_flat=pd.read_csv("df_test_flat.csv", index_col=0, parse_dates=True,
    squeeze=True)

df_test_reconstructed_flat=pd.read_csv("df_test_reconstructed_flat.csv",
    index_col=0, parse_dates=True, squeeze=True)
```

```
#Example for January
df_test_flat=df_test_flat[0:744] #January

anomalias_true = df_test_flat[df_test_flat.anomaly == True]

df_test_reconstructed_flat=df_test_reconstructed_flat[0:744]   #January


#Plot price
plt.plot(
  df_test_flat.index,
  df_test_flat.price,
  label='price'
);

#plot reconstructed price
plt.plot(
  df_test_reconstructed_flat.index,
  df_test_reconstructed_flat,
  label='reconstructed price'
);


#plot anomalies
sns.scatterplot(
  anomalias_true.index,
  anomalias_true.price,
  color=sns.color_palette()[3],
  s=52,
  label='anomaly'
)
plt.ylabel('price')
plt.legend();
```

B.4.2 Code for supply and demand graphs

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
pd.options.mode.chained_assignment = None

####
# Code for this section is only a slight variation of the code used for
# graphs in Power market and pricing section.
####


#Data for first graph
df=pd.read_csv("lockdown.csv", sep=",")

ofertada = df.loc[df["ofertada_casada"] == "O"]
casada = df.loc[df["ofertada_casada"] == "C"]

demanda_normal = ofertada.loc[ofertada["compra_venta"] == "C"]
oferta_normal = ofertada.loc[ofertada["compra_venta"] == "V"]

demanda_casada = casada.loc[casada["compra_venta"] == "C"]
oferta_casada = casada.loc[casada["compra_venta"] == "V"]


def acumulada(tipo):
    tipo=tipo.reset_index()
    cantidad = tipo["cantidad"]
```

```
    cant_acu = list()
    for i in range(len(cantidad)):
        if i == 0:
            cant_acu.append(cantidad[i])
        if i > 0:
            cant_acu.append(cantidad[i] + cant_acu[i - 1])

    tipo["cantidad_acumulada"] = cant_acu
    return tipo


demanda_normal = acumulada(demanda_normal)

oferta_normal = acumulada(oferta_normal)

demanda_casada = acumulada(demanda_casada)

oferta_casada = acumulada(oferta_casada)


fig, (ax1, ax2) = plt.subplots(1, 2)

ax1.plot(demanda_normal["cantidad_acumulada"], demanda_normal["precio"])
ax1.plot(oferta_normal["cantidad_acumulada"], oferta_normal["precio"])
ax1.plot(demanda_casada["cantidad_acumulada"], demanda_casada["precio"])
ax1.plot(oferta_casada["cantidad_acumulada"], oferta_casada["precio"])
ax1.legend(['demand', 'supply',"matched demand", "matched supply"], loc= "lower
     right")
ax1.title.set_text('02-16-2020 16:00')
ax1.grid(linestyle = '--', linewidth = 0.5)
ax1.set_xlim([0,60000])



#Data for second graph
df=pd.read_csv("lockdown-24.csv", sep=",")


ofertada = df.loc[df["ofertada_casada"] == "O"]
casada = df.loc[df["ofertada_casada"] == "C"]

demanda_normal = ofertada.loc[ofertada["compra_venta"] == "C"]
oferta_normal = ofertada.loc[ofertada["compra_venta"] == "V"]

demanda_casada = casada.loc[casada["compra_venta"] == "C"]
oferta_casada = casada.loc[casada["compra_venta"] == "V"]


def acumulada(tipo):
    tipo=tipo.reset_index()
    cantidad = tipo["cantidad"]
    cant_acu = list()
    for i in range(len(cantidad)):
        if i == 0:
            cant_acu.append(cantidad[i])
        if i > 0:
            cant_acu.append(cantidad[i] + cant_acu[i - 1])

    tipo["cantidad_acumulada"] = cant_acu
    return tipo


demanda_normal = acumulada(demanda_normal)
```

```
oferta_normal = acumulada(oferta_normal)

demanda_casada = acumulada(demanda_casada)

oferta_casada = acumulada(oferta_casada)


ax2.plot(demanda_normal["cantidad_acumulada"], demanda_normal["precio"])
ax2.plot(oferta_normal["cantidad_acumulada"], oferta_normal["precio"])
ax2.plot(demanda_casada["cantidad_acumulada"], demanda_casada["precio"])
ax2.plot(oferta_casada["cantidad_acumulada"], oferta_casada["precio"])
ax2.legend(['demand', 'supply',"matched demand", "matched supply"], loc= "lower
    right")
ax2.title.set_text('02-15-2020 16:00')
ax2.grid(linestyle = '--', linewidth = 0.5)
ax2.set_xlim([0,60000])
```

C R code for ARIMA modeling

C.1 R code for ARIMA model

```
library(tfarima)
library(Metrics)


df <- read.table("precios_14_20.txt", header = T, sep=",", col.names = c("date"
    , "price"))
rownames(df) <- df$date
df$date <- NULL


# June 2019 - December 2019
Y <- df[47449:52584,]

# 2020
test <- df[52585:61368,]



#IDENTIFICATION



#Range-median graph
ide(Y, transf = list(list(), list(bc = T)), graphs = c("plot", "rm") )

#Barlett
n <- length(Y)
nj <- 48
k <- n %/% nj
g <- rep(1:k, each = nj)
g <- g[1:n]
length(g)


h1.test <- bartlett.test(Y, g)
h2.test <- bartlett.test(log(Y), g)
h1.test
h2.test
```

```
#ACF and PACF
ide(Y, graphs = c ("plot", "acf", "pacf"), lag.max = 216, lags.at = 24)


# Regular difference
ide(Y, transf = list(d = 1), lag.max = 216, lags.at = c(24, 168))

# 1, 24 and 168 differences
i <- um(i = "(1-B1)(1-B24)(1-B168)")$i
ide(Y, transf = list(i = i), lag.max = 216, lags.at = c(24, 168))


ma <- um(ma = "(1 - 0.6B24)(1 - 0.6B168)")
display(list(ma), graphs = c("acf", "pacf"), lag.max = 400, byrow = T)



#ESTIMATION AND DIAGNOSTIC CHECKING


#ARIMA modeling with a short time series
Y1 <- ts(window(Y, end = 24*7*8), start = c(1,1), frequency = 24)


um1 <- um(Y1, ar = "(1-0.15B)(1-0.5B24)(1-0.5B168)", i = list(1), ma = "(1-0.1
    B24)(1-0.1B168)")
summary(um1)

diagchk(um1, lag.max = 168*3, lags.at = c(24, 168))



um2 <- modify(um1, ar = "(1:12)/24")
um2
summary(um2)
diagchk(um2, lag.max = 168*3, lags.at = c(24, 168))


#Fitting the ARIMA model to the long time series
Y <- ts(Y, start = c(1, 1), frequency = 24)
um3 <- fit(um2, Y)
summary(um3)

diagchk(um3)



#PREDICT

n <- length(Y)
Yf <- ts(df[47449:61368,]) #all data

predictions <- sapply(1:366, function(x) {
  ori <- n + (x-1)*24
  p <- predict(um3, Yf, ori = ori, n.ahead = 24)
  p$z[(ori+1):(ori+24)]
})


write.csv(predictions,"ARIMA_PRED.csv", row.names = FALSE)
```

C.2 R code for Transfer Function Model

```
library(tfarima)
```

```
# Read data
readtfg <- function(url, name){
  df <- read.table(url,
                   header = T, sep=",", col.names = c("date", name))
  rownames(df) <- df$date
  df$date <- NULL
  return(df)
}

df1 <- readtfg("precios_14_20.txt", "price")
df2 <- readtfg("eolica_14_20.csv", "wind")
df3 <- readtfg("demanda_14_20.csv", "demand")

demand <- ts(df3[47449:52584,]) #June - December
demand_test <- ts(df3[52585:61368,])#2020

wind <- ts(df2[47449:52584,])
wind_test <- ts(df2[52585:61368,])

price <- ts(df1[47449:52584,])
price_test <- ts(df1[52585:61368,])


# Demand

umx1 <- um(demand, ar = list(1, "(1:12)/24"), i = list(1, c(1, 24), c(1, 168)),
     ma = "(1 - 0.1B24)(1-0.1B168)")
umx1
summary(umx1)
umx1$ar

diagchk(umx1)


# Predictions for demand
n <- length(demand)
demand2 <- ts(df3[47449:61368,]) #train + test
demandf <- sapply(1:366, function(x) {
  ori <- n + (x-1)*24
  p <- predict(umx1, demand2, ori = ori, n.ahead = 24)
  p$z[(ori+1):(ori+24)]
})
save(demandf, file = "demandf.Rda")
load("demandf.Rda")



# Wind

ide(wind, transf = list(d = 1))
umx2 <- um(wind, ar = list("(1-0.5B)", "(1:12)/24"), i = list(1, c(1, 24)), ma
    = "(1 - 0.1B24)")
umx2
summary(umx2)
umx2$ar
diagchk(umx2)


# Predictions for wind
n <- length(wind)
wind2 <- ts(df2[47449:61368,])
```

```
windf <- sapply(1:366, function(x) {
  ori <- n + (x-1)*24
  p <- predict(umx2, wind2, ori = ori, n.ahead = 24)
  p$z[(ori+1):(ori+24)]
})
windf
save(windf, file = "windf.Rda")
load("windf.Rda")


# TRANSFER FUNCTION MODEL

# Univariate model
umy <- um(price, ar = list(1, "(1:12)/24"), i = list(1, c(1, 24), c(1, 168)),
    ma = "(1 - 0.8B24)(1-0.7B168)")


y  <- residuals(umy)
x1 <- residuals(umx1)
x2 <- residuals(umx2)


# Cross-correlation functions
pccf(x1, y)
pccf(x2, y)
pccf(x1, x2)


X1 <- c(demand, as.vector(demandf))
tfx1 <- tf(X1, w0 = 0.0012)


X2 <- c(wind, as.vector(windf))
tfx2 <- tf(X2, w0 = -0.0013, ar = "1 - 0.3B")



tfm1 <- tfm(inputs = list(tfx1, tfx2), noise = umy)
u <- residuals(tfm1)

diagchk(tfm1, lags.at= c(24, 168))

umu <- um(u, ma = c(2, 168))
diagchk(umu, lags.at= 24)


tfm2 <- modify(tfm1, ma = "1 - 0.1447080B168-0.1222818B336")


diagchk(tfm2, lags.at= 24)


# Predict
n <- length(price)
Yf <- ts(df1[47449:61368,])
predictions <- sapply(1:366, function(x) {
  ori <- n + (x-1)*24
  p <- predict(tfm2, y = Yf, ori = ori, n.ahead = 24)
  p$z[(ori+1):(ori+24)]
})
```

```
length(price_test)
length(predictions)


write.csv(predictions,"TF.csv", row.names = FALSE)
```

D Python code for Neural Networks modeling

D.1 Python code for Recursive Multi-step Neural Network

```
import numpy as np
import pandas as pd
from tensorflow import keras
import tensorflow as tf
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt

#split data
def split_dataset(data):
  train, test = data[:-8760], data[-8760:]
  train = np.array(train)
  test = np.array(test)
  return train, test

#data to supervised learning
def to_supervised(train, n_input, n_out=1):
    X, y = list(), list()
    in_start = 0
    for _ in range(len(train)):
        in_end = in_start + n_input
        out_end = in_end[2] + n_out
        if out_end <= len(train):
            x_input = np.stack((np.array(train[(in_end[2]-n_input[0]):(out_end-
    n_input[0]), 0]),
                                np.array(train[(in_end[2]-n_input[1]):(out_end-
    n_input[1]), 0]),
                                np.array(train[(in_end[2]-n_input[2]):(out_end-
    n_input[2]), 0])),axis=1 )
            X.append(x_input)
            y.append(train[in_end[2]:out_end, 0])
        in_start += 1
    return np.array(X), np.array(y)

# model
def RNN():
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2],
    train_y.shape[1]
  model = keras.Sequential()
  model.add(keras.layers.LSTM(10, activation='relu', input_shape=(n_timesteps,
    n_features)))
  model.add(keras.layers.Dense(4, activation='relu'))
  model.add(keras.layers.Dense(n_outputs))
  opt=keras.optimizers.Adam(learning_rate=0.001)
  model.compile(loss='mae', optimizer=opt)
  history=model.fit(train_x, train_y, epochs=15, batch_size=30, verbose=1,
    callbacks=[callback,callback2])
  return model, history


# Learning rate schedule
def scheduler(epoch, lr):
  if epoch < 10:
    return lr
```

```
  if epoch < 20:
    return 0.0005
  else:
      return 0.0001


callback = tf.keras.callbacks.LearningRateScheduler(scheduler)

# Early stopping
callback2 = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=6,
    verbose=1)

# Predictions:
# We add one by one the predictions to data until the end of the day (24
    predictions),
# then we replace these predictions by the real values (from the test).
def predictions(n_input):
  data=train
  j=1
  predicciones = np.empty((0,0), int)
  for i in range(len(test)):
    k = len(train) + 24 * j
    test_x = np.stack((np.array(data[-n_input[0], 0]),
                           np.array(data[-n_input[1], 0]),
                           np.array(data[-n_input[2], 0])))
    test_x=test_x.reshape(1,1,3)
    if len(data) < k:
      yhat = model.predict(test_x)
      predicciones=np.append(predicciones,yhat)
      data=np.append(data, yhat, axis=0)
    if len(data) == k:
      data = data[:-24]
      new_test = test[24*(j-1) : 24*j]
      data=np.append(data, new_test, axis=0)
      j += 1
  return predicciones



df = pd.read_csv("precios_luz_14_19.txt", sep=",", index_col=0)

train, test = split_dataset(df)

#Lags selection
n_input = np.array([1,24,168])

train_x, train_y = to_supervised(train, n_input)

model, history = RNN()

#plot mae per epoch
plt.plot(history.history["loss"])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['loss', 'val_loss'], loc='upper right')
plt.show()


pred = predictions(n_input)

mae=mean_absolute_error(test, pred)
```

D.2 Python code for hyperparameter optimization with keras tuner

```
import numpy as np
import pandas as pd

import tensorflow as tf
from tensorflow import keras
import kerastuner as kt



df=pd.read_csv("precios_14_20.txt", sep=",", index_col=0)

# split data
def split_dataset(data):
  train, test = data[:-17544], data[-17712:-8784] # test 2019
  train = np.array(np.split(train, len(train)/24))
  test = np.array(np.split(test, len(test)/24))
  return train, test



#data to supervised learning
def to_supervised_3(set, n_input, n_out=24):
  data = set.reshape((set.shape[0]*set.shape[1], set.shape[2]))
  X, y = list(), list()
  in_start = 0
  for _ in range(len(data)):
    in_end = in_start + n_input
    out_end = in_end[2] + n_out
    if out_end <= len(data):
      x_input = np.stack((np.array(data[(in_end[2] - n_input[0]):(out_end -
    n_input[0]), 0]),
                np.array(data[(in_end[2] - n_input[1]):(out_end - n_input[1]),
    0]),
                np.array(data[(in_end[2] - n_input[2]):(out_end - n_input[2]),
    0])), axis=1)
      X.append(x_input)
      y.append(data[in_end[2]:out_end, 0])
    in_start += 24
  return np.array(X), np.array(y)

# train the model
def build_model(hp):
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2],
    train_y.shape[1]
  model = keras.Sequential()
  model.add(keras.layers.LSTM(units=hp.Int('units', min_value=50, max_value
    =500, step=50),input_shape=(n_timesteps, n_features), activation='relu'))
  model.add(keras.layers.Dense(units=hp.Int('units_dense', min_value=0,
    max_value=400, step=50), activation='relu'))
  model.add(keras.layers.Dense(n_outputs))
  model.compile(loss='mae', optimizer=keras.optimizers.Adam(hp.Choice('
    learning_rate', [0.001, 0.0001, 0.00001])))
  return model


train, test= split_dataset(df)

#set lags
n_input=np.array([24,144,168])

train_x, train_y = to_supervised_3(train, n_input)
```

```
val_x, val_y = to_supervised_3(test, n_input)

callback2 = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3,
    verbose=1)


tuneo = kt.RandomSearch(
    build_model,
    objective='val_loss',
    max_trials=30,
    executions_per_trial=2,
    directory='my_directory',
    project_name='tuner_RNN')


tuneo.search_space_summary()

tuneo.search(train_x, train_y, epochs = 40, batch_size=30, verbose=1, callbacks
    =[callback2], validation_data = (val_x, val_y))

tuneo.results_summary()
```

D.3 Python code for Neural Networks models (CNN, RNN and CNN-RNN)

```
import numpy as np
import pandas as pd
from numpy import split
from numpy import array

from tensorflow import keras
import tensorflow as tf
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.regularizers import l2
#from keras.utils.vis_utils import plot_model

from sklearn.preprocessing import MinMaxScaler



# Split data
def split_dataset(data):
  train, test = data[:-8784], data[-8952:]  # train 14-19, test 2020
  scaler = MinMaxScaler(feature_range=(-1, 1)) # rescale data [-1, 1]
  scaler = scaler.fit(train)
  train = scaler.transform(train)
  test = scaler.transform(test)
  train = array(split(train, len(train)/24))
  test = array(split(test, len(test)/24))
  return train, test, scaler

# Data to supervised learning
def to_supervised_3(set, n_input, n_out=24):
  data = set.reshape((set.shape[0]*set.shape[1], set.shape[2]))
  X, y = list(), list()
  in_start = 0
  for _ in range(len(data)):
    in_end = in_start + n_input
    out_end = in_end[2] + n_out
    if out_end <= len(data):
      x_input = np.stack((np.array(data[(in_end[2] - n_input[0]):(out_end -
    n_input[0]), 0]),
                np.array(data[(in_end[2] - n_input[1]):(out_end - n_input[1]),
    0]),
```

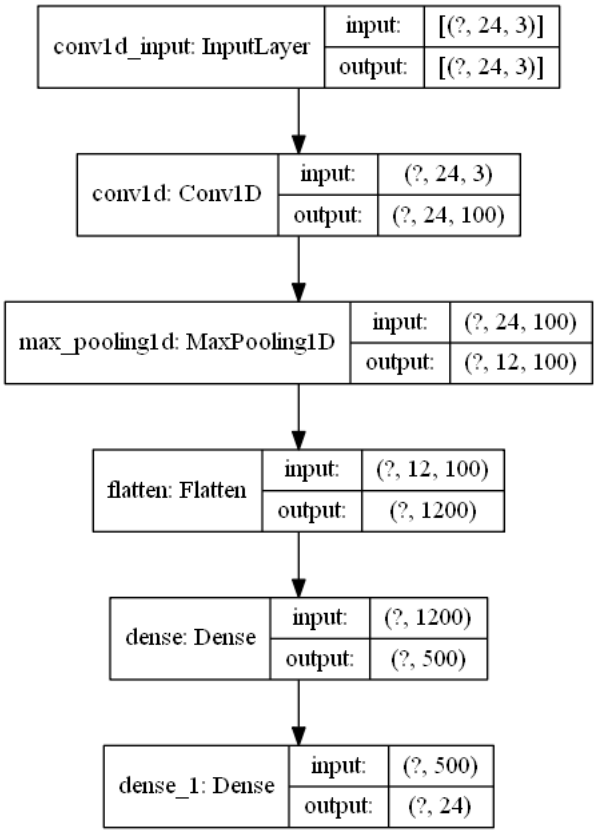
```
                np.array(data[(in_end[2] - n_input[2]):(out_end - n_input[2]),
    0])), axis=1)
      X.append(x_input)
      y.append(data[in_end[2]:out_end, 0])
    in_start += 24
  return array(X), array(y)


# Data to supervised (for exogenous variables)
def to_supervised(set, n_input, n_out=24):
  data = set.reshape((set.shape[0]*set.shape[1], set.shape[2]))
  X = list()
  in_start = 0
  for _ in range(len(data)):
    in_end = in_start + n_input
    out_end = in_end + n_out
    if out_end <= len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
    in_start += 24
  return array(X)

#CNN
def CNN(train_x_final, train_y):
  n_timesteps, n_features, n_outputs = train_x_final.shape[1], train_x_final.
    shape[2], train_y.shape[1]
  model = keras.Sequential()
  model.add(keras.layers.Conv1D(filters=100, kernel_size=4,kernel_regularizer=
    l2(0.0004),bias_regularizer=l2(0.0004),
                                  padding="same", activation='relu',
    input_shape=(n_timesteps, n_features)))
  model.add(keras.layers.MaxPooling1D(pool_size=2))
  model.add(keras.layers.Flatten())
  model.add(keras.layers.Dense(500, activation='relu', kernel_regularizer=l2
    (0.0004), bias_regularizer=l2(0.0004)))
  model.add(keras.layers.Dense(n_outputs))
# tf.keras.utils.plot_model(model, to_file='CNN.png', show_shapes=True,
    show_layer_names=True)
  opt=keras.optimizers.Adam(learning_rate=0.001)
  model.compile(loss='mae', optimizer=opt)
  model.fit(train_x_final, train_y, epochs=40, batch_size=30, verbose=1,
    callbacks=[callback, callback2])
  return model

#RNN
def RNN(train_x_final, train_y):
  n_timesteps, n_features, n_outputs = train_x_final.shape[1], train_x_final.
    shape[2], train_y.shape[1]
  model = keras.Sequential()
  model.add(keras.layers.GRU(300, activation='relu', input_shape=(n_timesteps,
    n_features),
                kernel_regularizer=l2(0.0002),bias_regularizer=l2(0.0002)))
  model.add(keras.layers.Dense(300, activation='relu', kernel_regularizer=l2
    (0.0002),bias_regularizer=l2(0.0002)))
  model.add(keras.layers.Dense(n_outputs))
# tf.keras.utils.plot_model(model, to_file='RNN.png', show_shapes=True,
    show_layer_names=True)
  opt=keras.optimizers.Adam(learning_rate=0.001)
  model.compile(loss='mae', optimizer=opt)
  model.fit(train_x_final, train_y, epochs=40, batch_size=30, verbose=1,
    callbacks=[callback, callback2])
  return model
```

```
# CNN and RNN
def Hibrid(train_x_final, train_y):
  n_timesteps, n_features, n_outputs = train_x_final.shape[1], train_x_final.
    shape[2], train_y.shape[1]
  model = keras.Sequential()
  model.add(keras.layers.Conv1D(filters=100, kernel_size=4, kernel_regularizer=
    l2(0.0004),bias_regularizer=l2(0.0004),
                            padding="same", activation='relu', input_shape=(
    n_timesteps, n_features)))
  model.add(keras.layers.MaxPooling1D(pool_size=2))
  model.add(keras.layers.Flatten())
  model.add(keras.layers.RepeatVector(n_outputs))
  model.add(keras.layers.GRU(300, activation='relu', return_sequences=True,
                kernel_regularizer=l2(0.0002), bias_regularizer=l2(0.0002)
    ))
  model.add(keras.layers.TimeDistributed(keras.layers.Dense(300, activation='
    relu',
                kernel_regularizer=l2(0.0004), bias_regularizer=l2(0.0004)
    )))
  model.add(keras.layers.TimeDistributed(keras.layers.Dense(1)))
# tf.keras.utils.plot_model(model, to_file='CNNandRNN.png', show_shapes=True,
    show_layer_names=True)
  opt=keras.optimizers.Adam(learning_rate=0.001)
  model.compile(loss='mae', optimizer=opt)
  model.fit(train_x_final, train_y, epochs=40, batch_size=30, verbose=1,
    callbacks=[callback, callback2])
  return model


# Learning rate schedule
def scheduler(epoch, lr):
  if epoch < 10:
    return lr
  if epoch < 20:
    return 0.0005
  else:
      return 0.0001

# Early stopping
callback2 = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=5,
    verbose=1)

callback = tf.keras.callbacks.LearningRateScheduler(scheduler)


# Build model, predict and retrain
def built_predict(build_model):
  history_x = train_x_final
  history_y = train_y
  predictions = np.empty((0,0), int)

  for i in range(test_y.shape[0]):
    if i % 92 == 0 : #quadrimestral
      model = build_model(history_x, history_y)
    t_x = test_x_final[i]
    t_x = t_x.reshape(1, t_x.shape[0], t_x.shape[1])
    history_x = np.append(history_x, t_x, axis=0)
    last = history_x[-1,]
    last = last.reshape(1, last.shape[0], last.shape[1])
    yhat=model.predict(last)
    predictions = np.append(predictions, yhat)
```

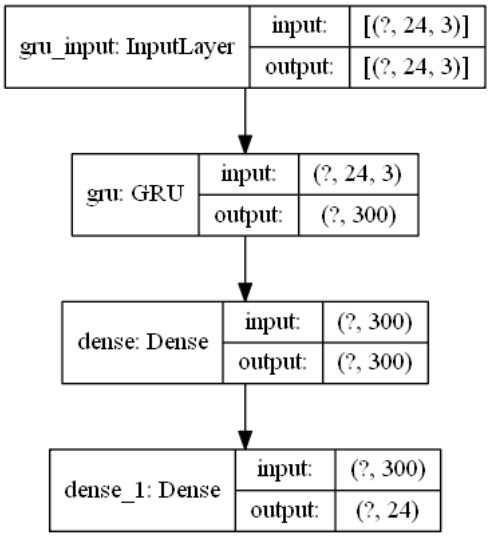
```
    t_y = test_y[i]
    t_y = t_y.reshape(1, t_y.shape[0])
    history_y = np.append(history_y, t_y, axis=0)

  predictions = predictions.reshape(int(predictions.shape[0]/24), 24)
  predictions = scaler1.inverse_transform(predictions)
  return predictions


# Read data
df1=pd.read_csv("precios_14_20.txt", sep=",", index_col=0)
df2=pd.read_csv("eolica_14_20.csv", sep=",", index_col=0)
df3=pd.read_csv("demanda_14_20.csv", sep=",", index_col=0)



train, test, scaler1= split_dataset(df1)

# Lags
n_input=array([24,144,168])


train_x, train_y = to_supervised_3(train, n_input)
test_x, test_y = to_supervised_3(test, n_input)


# Exogenous variables processing
def multi(df):
    train, test, scaler = split_dataset(df)
    input = 24
    train_x = to_supervised(train, input)
    test_x = to_supervised(test, input)
    train_x = train_x[int(n_input[2] / 24 - 1):]
    test_x = test_x[int(n_input[2] / 24 - 1):]
    return train_x, test_x

train_x_df2, test_x_df2 =multi(df2)
train_x_df3, test_x_df3 =multi(df3)


#Final data

# Multivariate model
#train_x_final=np.dstack((train_x,train_x_df2, train_x_df3))
#test_x_final=np.dstack((test_x,test_x_df2, test_x_df3))

# Univariante model
train_x_final=train_x
test_x_final=test_x


# Train model "n_members" times and save predictions
n_members = 10
for i in range(n_members):
  yhats = built_predict(CNN)
  filename = "predictions/um_CNN" + str(i + 1)
  np.save(filename, yhats)
  print('Saved: %s' % filename)
```

E Neural Networks architectures diagrams (for univariate models)

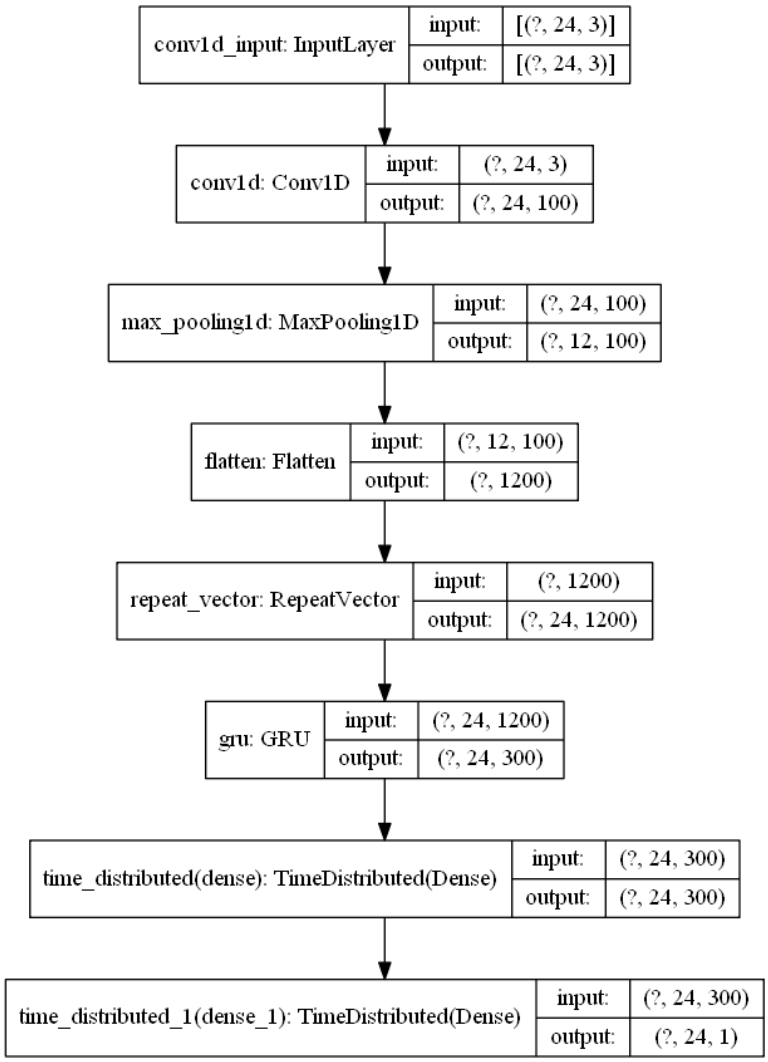
E.1 CNN diagram



E.2 RNN diagram



E.3 CNN-RNN diagram



F Python code for ARIMA and NN ensembling

```
import numpy as np
import pandas as pd


# Load CNN predictions
pred_cnn = np.load("mmcnn.npy")
pred_cnn = pred_cnn.reshape(pred_cnn.shape[0]*pred_cnn.shape[1])

# Load ARIMA predictions
pred_arima = pd.read_csv("ARIMA_FINAL.csv")
pred_arima = pred_arima.to_numpy()
pred_arima = pred_arima.transpose()
pred_arima = pred_arima.reshape(pred_arima.shape[0]*pred_arima.shape[1],)

#SIMPLE AVERAGE

predictions = (pred_cnn*0.5 + pred_arima*0.5)

# Save
np.save("ARIMA-CNN.npy", predictions)
```

G Python code for results section

G.1 Python code for Naive model

```
import numpy as np
import pandas as pd
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_percentage_error

#Naive model where predictions for day t are data values for day t-1

df=pd.read_csv("precios_14_20.txt", sep=",", index_col=0)


def split_data(data):
  train, test = data[0:-8784], data[-8784:]
  train = np.array(np.split(train, len(train)/24))
  test = np.array(np.split(test, len(test)/24))
  return train, test

train, test= split_data(df)

a=train[-1,] #last 24h of train set
a=a.reshape(a.shape[0])

b=test[0:-1] #2020 without last 24h
b=b.reshape(b.shape[0]*b.shape[1])


predicted=np.concatenate((a,b))

actual= test.reshape(test.shape[0]*test.shape[1])

mae = mean_absolute_error(actual, predicted)
rmse = np.sqrt(mean_squared_error(actual, predicted))
mape = mean_absolute_percentage_error(actual, predicted)*100
```

G.2 Python code for tables and barplots

```
import numpy as np
import pandas as pd
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
pd.options.mode.chained_assignment = None
from datetime import datetime


# Load test data
test = np.load("test2020.npy")
test = test.reshape(test.shape[0]*test.shape[1])

# Load NN univariate o multivariate
pred_cnn = np.load("mmcnn.npy")
pred_cnn = pred_cnn.reshape(pred_cnn.shape[0]*pred_cnn.shape[1])

pred_rnn = np.load("mmrnn.npy")
pred_rnn = pred_rnn.reshape(pred_rnn.shape[0]*pred_rnn.shape[1])

pred_crnn = np.load("mmcnnandrnn.npy")
pred_crnn = pred_crnn.reshape(pred_crnn.shape[0]*pred_crnn.shape[1])

# Load ARIMA or TFM
```

```
pred_arima = pd.read_csv("ARIMA_FINAL.csv")
pred_arima = pred_arima.to_numpy()
pred_arima = pred_arima.transpose()
pred_arima = pred_arima.reshape(pred_arima.shape[0]*pred_arima.shape[1],)



#### CREATE DATABASE FOR TABLES

#MAE
mae_cnn = mean_absolute_error(test, pred_cnn)
mae_rnn = mean_absolute_error(test, pred_rnn)
mae_arima = mean_absolute_error(test, pred_arima)
mae_crnn = mean_absolute_error(test, pred_crnn)


#RMSE
def rmse(predictions):
    rmse = np.sqrt(sum((test-predictions)**2)  / len(test))
    return rmse

rmse_cnn = rmse(pred_cnn)
rmse_rnn = rmse(pred_rnn)
rmse_arima = rmse(pred_arima)
rmse_crnn = rmse(pred_crnn)

#MAPE
def mape(predictions):
    mape = np.mean(np.abs((test - predictions)/test))*100
    return mape

mape_cnn = mape(pred_cnn)
mape_rnn = mape(pred_rnn)
mape_arima = mape(pred_arima)
mape_crnn = mape(pred_crnn)

# Create dataframe

df_error=pd.DataFrame(data={"RMSE": [rmse_arima, rmse_cnn, rmse_rnn, rmse_crnn
    ],
                            "MAE": [mae_arima, mae_cnn, mae_rnn, mae_crnn],
                            "MAPE": [mape_arima, mape_cnn, mape_rnn, mape_crnn
    ]}, index = ["ARIMA", "CNN", "RNN", "CNNandRNN"])




#### CREATE DATABASE FOR BARPLOTS

# Database. Columns: Error, day, hour, month
def data_base(error):
    df1=pd.read_csv("precios_14_20.txt", sep=",", index_col=0)
    df=df1[-8784:]
    error = error.tolist()
    df["error"]=error
    df=df.drop("price", axis = 1)
    index = df.index
    day = list()
    hour = list()
    month = list()
    for i in index:
        day.append(i[0:10])
```

```
        hour.append(i[11:13])
        month.append(i[5:7])
    day_dt = [datetime.strptime(x, '%Y-%m-%d') for x in day]
    weekday = [x.weekday() for x in day_dt]
    df["day"] = weekday
    df["hour"] = hour
    df["month"] = month
    return df


# MSE for each of the hours
def mse_hours(predictions):
    mse = (test- predictions)**2
    return mse

mse_hours_cnn =   mse_hours(pred_cnn)
mse_hours_rnn =   mse_hours(pred_rnn)
mse_hours_arima = mse_hours(pred_arima)
mse_hours_crnn =  mse_hours(pred_crnn)


df_cnn = data_base(mse_hours_cnn)
df_rnn = data_base(mse_hours_rnn)
df_arima = data_base(mse_hours_arima)
df_crnn = data_base(mse_hours_crnn)


# Group data by day, hour and month. Output: RMSE per day, hour and month
def group_data(df):
    day_error=df.groupby(["day"])["error"].mean()
    hour_error=df.groupby(["hour"])["error"].mean()
    month_error=df.groupby(["month"])["error"].mean()
    day_error = day_error.tolist()
    hour_error = hour_error.tolist()
    month_error = month_error.tolist()
    return np.sqrt(day_error), np.sqrt(hour_error), np.sqrt(month_error)


day_cnn, hour_cnn, month_cnn = group_data(df_cnn)
day_ARIMA, hour_ARIMA, month_ARIMA = group_data(df_arima)
day_rnn, hour_rnn, month_rnn = group_data(df_rnn)
day_crnn, hour_crnn, month_crnn = group_data(df_crnn)




# Hours NEURAL NETWORKS
x = np.arange(24)
width = 0.20
x_labels = ["00", "01", "02","03", "04", "05", "06","07", "08", "09","10","11",
    "12","13","14","15","16","17","18","19",
      "20","21","22","23"]

plt.bar(x -0.2, hour_cnn, width, color = "SkyBlue" , edgecolor="black")
plt.bar(x , hour_rnn, width, edgecolor="black", color = "IndianRed")
plt.bar(x + 0.2 , hour_crnn, width, edgecolor="black", color = "forestgreen")
plt.title("RMSE per hours")
plt.ylabel("RMSE")
plt.xticks(x, x_labels, rotation=0)
plt.legend(["CNN", "RNN", "CNN+RNN"])
plt.show()
```

```
# Hours NN VS ARIMA
x = np.arange(24)
width = 0.40
x_labels = ["00", "01", "02","03", "04", "05", "06","07", "08", "09","10","11",
    "12","13","14","15","16","17","18","19",
      "20","21","22","23"]

plt.bar(x - 0.2, hour_cnn, width, color = "SkyBlue" , edgecolor="black")
plt.bar(x + 0.2, hour_ARIMA, width, color = "IndianRed", edgecolor="black")
plt.title("RMSE per hours")
plt.ylabel("RMSE")
plt.xticks(x, x_labels, rotation=0)
plt.legend(["CNN", "ARIMA"])
plt.show()
```

```
### Days
x = np.arange(7)
width = 0.40
x_labels = ["Monday", "Tuesday", "Wednesday", "Thrusday", "Friday", "Saturday",
     "Sunday"]

plt.bar(x - 0.2, day_cnn, width, color = "SkyBlue" , edgecolor="black")
plt.bar(x + 0.2, day_ARIMA, width, color = "IndianRed", edgecolor="black")
plt.title("RMSE per days")
plt.ylabel("RMSE")
plt.xticks(x, x_labels, rotation=45)
plt.legend(["CNN", "ARIMA"])
plt.show()
```

```
### Months
x = np.arange(12)
width = 0.40
x_labels = ["January","February","March","April","May","June","July","August","
    September","October","November",
            "December"]
plt.bar(x - 0.2, month_cnn, width, color = "SkyBlue" , edgecolor="black")
plt.bar(x + 0.2, month_ARIMA, width, color = "IndianRed", edgecolor="black")
plt.title("RMSE per months")
plt.ylabel("RMSE")
plt.xticks(x, x_labels, rotation=45)
plt.legend(["CNN", "ARIMA"])
plt.show()
```

G.3 Python code for kernels graph

```
import matplotlib.gridspec as gridspec
from tensorflow import keras
import matplotlib.pyplot as plt

# Load model
model = keras.models.load_model("umCNN.h5")

filters, biases = model.layers[0].get_weights()

# normalize filter values to 0-1
f_min, f_max = filters.min(), filters.max()
```

```
filters = (filters - f_min) / (f_max - f_min)

# Plot
plt.figure(figsize = (10,10))
gs1 = gridspec.GridSpec(10, 10)
gs1.update(wspace=0.025, hspace=0.15) # set the spacing between axes.

for i in range(100):
  f = filters[:, :, i]
  ax1 = plt.subplot(gs1[i])
  plt.imshow(f)
  plt.axis('off')
  ax1.set_xticklabels([])
  ax1.set_yticklabels([])
  ax1.set_aspect('equal')

plt.show()
```