Universidad de Valladolid

Grado en Ingeniería Informática

Final Degree Project

Energy Load Forecast in Smart Buildings with Deep Learning Techniques

Author

Alejandro Barón García

Supervisors

Carlos Javier Alonso González

José Belarmino Pulido Junquera

ACADEMIC YEAR

2019-2020

Abstract

Predicting energy load is a growing problem these days. The need to study in advance how electricity consumption will behave is key to resource management.

Especially interesting is the case of the so-called Smart Buildings, buildings born from the trend towards sustainable development and consumption which is increasingly in vogue, becoming mandatory by law in many countries.

One type of model that constitutes an important part of the state of the art are the models based on Deep Learning. These models represented great advances in Artificial Intelligence recently, since although they were born in the 20th century, it has not been until 10 years ago that they have re-emerged thanks to the computational advances that allow them to be trained by the general public.

In this Final Degree Project, advanced Deep Learning techniques applied to the problem of load prediction in Smart Buildings are presented, mainly basing the development on the data from the Alice Perry building of the National University of Ireland Galway, in collaboration with the Informatics Research Unit for Sustainable Engineering of the same university.

The datasets used were obtained from the time series of aggregated electricity consumption of the air handling units (AHUs) in the Alice Perry building. Along with this information, historical weather data were also collected from the weather station in the same building in order to study if these climatic variables help to a better prediction in the models.

Time series prediction on this energy load data will be made in two different ways with hourly granularity: one-step prediction in which studying the previous observations an estimate of the value of the load in the next hour is obtained and sequence prediction, in which we will try to predict the behaviour of the series in the next hours from the previous values.

Resumen

La predicción de carga energética es un problema al alza actualmente. La necesidad de estudiar con antelación cómo se va a comportar el consumo eléctrico es clave para la gestión de recursos.

Especialmente interesante es el caso de los llamados Smart Buildings, edificios nacidos por la tendencia hacia un desarrollo y consumo sostenible el cual cada vez está más en boga, llegando a ser obligatorio por ley en muchos países.

Un tipo de modelos que constituyen una parte importante del estado del arte son los modelos basados en Deep Learning. Estos modelos supusieron grandes avances en la Inteligencia Artificial recientemente, ya que aunque nacidos en el Siglo XX, no ha sido hasta escasos 10 años cuando han resurgido gracias a los avances computacionales que permiten entrenarlos por el público general.

En este trabajo de fin de grado se presentan técnicas avanzadas de Deep Learning aplicadas al problema de la predicción de carga en Smart Buildings, principalmente basando el desarrollo en los datos del edificio Alice Perry de la National University of Ireland Galway, en colaboración con el grupo Informatics Research Unit for Sustainable Engineering de la misma universidad.

Los conjuntos de datos utilizados se obtuvieron datos sobre la serie temporal de consumo eléctrico agregado de los aires acondicionados en el edificio Alice Perry. Junto a esta información, se recopilaron también datos meteorológicos históricos de la estación meteorológica en el mismo edificiocon el objetivo de estudiar si estas variables climáticas ayudan a una mejor predicción en los modelos.

La predicción de series temporales sobre estos datos de carga energética se realizará en dos modos con granularidad horaria: La predicción a un paso en la que estudiando las observaciones anteriores se obtiene una estimación del valor de la carga en la próxima hora y predicción de secuencias, en la que se intentará predecir el comportamiento de la serie en las próximas horas a partir de los valores anteriores.

For life is quite absurd
And death's the final word
You must always face the curtain with a bow
Forget about your sin
Give the audience a grin
Enjoy it, it's your last chance anyhow

Always look on the bright side of life - Eric Idle, Monty Python

Acknowledgements

To my two supervisors, Carlos & Belarmino for the humongous effort and help that they put into this final degree project and their trust in my work.

To Desirée for helping me in Galway and helping me getting into the research scene.

To the IRUSE group, I want to thank everyone specially Marcus Keane for accepting me, guiding me and treating me like another member of the group.

To my friends, who whenever I felt down or needed to get away, they were always there to cheer me up, specially in these last hard months. Some of you have been with me since we were 6, some of you we started university and some of you just came a couple years ago when sport brought us together. It's been (and will be) a pleasure to walk with you by my side.

Last, but not least, to my parents, Edith & Juan Luis for helping me getting through these difficult times during my university degree. You have helped me to push through my studies and wouldn't have been able to complete them without your encouragement. I love you.

Contents

1. Introduction

1.1 Motivation

Wang et al. [1] state that "*In recent years, the power industry has witnessed considerable developments of data analytics in the processes of generation, transmission, equipment, and consumption*". They also showed that number of publications growed from 2010 to 2017, rapidly increasing from 2012. The availability of data, smart meters and the development of new Machine Learning techniques made it possible to create data-driven models for the energy consumption field.

From the perspective of the Degree in Computer Engineering, the interest relies in continuing learning the Deep Learning techniques introduced in the *Machine Learning Techniques* and *Data Mining* subjects, deepening the baseline knowledge taught in them. This final degree project will be a first step into the research scene, motivating a literature review that will help extend and strengthen not only the domain knowledge gained while doing the project but researching competences.

The main idea behind the definition of this project is to study the performance of different deep learning architectures for time series forecast, more specifically applied to energy load forecast in smart buildings.

1.2 Problem Classification

In their review, Hamad et al. [2] propose four classes attending to the range of the load forecast:

- **Long-term load forecasting** (LTLF): for one year or more ahead, usually carried by strategic planning projects.
- **Medium-term load forecasting** (MTLF): from one week to a year, this range is mainly used for applications like maintenance schedule and fuel purchase planning.

- **Short-term load forecasting** (STLF): from one hour to one week, allowing to carry day-to-day applications.
- **Ultra/very short-term load forecasting** (VSTLF): from minutes to an hour ahead, usually needed for real-time control.

According to Lin et al. ([3] trough [2]) there are two main families of models for load prediction

- **Multi-factor forecasting methods**: which focus on creating models that portray the relationship between significant factors and the energy load in the system.
- **Time series forecasting methods**: which focus on using existing historical data to build data-driven models. Within these models we can find three main families:
  - *Statistical models* (SM): These methods assume independence and normality of the residuals (there is an underlying probability distribution in the whole process). This forces strong assumptions on the data which can result in a weaker predictive power, but have great interpretability
  - *Machine learning models* (ML) : Which do not assume any probability distribution, thus their predictive power can be better than SMs. However, these methods can lack interpretability (i.e. ensemble methods, neural networks).
  - *Hybrid models* (HM): These methods present a mixture of the above mentioned, using them together to model the data.

This Final Degree Project focuses on ML models, more precisely Deep Learning (DL) models applied to short-term forecasting.

2. Planning

2.1 Objectives

The aim of this project is the revision of the existing techniques of Deep Learning for the prediction of time series in the domain of short and medium term prediction of energy demand in buildings.

2.1.1 Tasks

In order to achieve this objective, the following tasks will be developed:

- Initial study of Deep Learning, fundamentals and operation of the basic techniques that make up the advanced models.
- Study of existing literature and proposals for advanced neural network architectures for time series forecasting.
- Validate the proposed techniques applied in real word data, the Alice Perry Building and a public dataset.
    - Data pre-processing and cleaning to enable Alice Perry data use in the problem of time series prediction.
    - Training and testing advanced deep learning architectures on these datasets

2.2 Research Process Planning

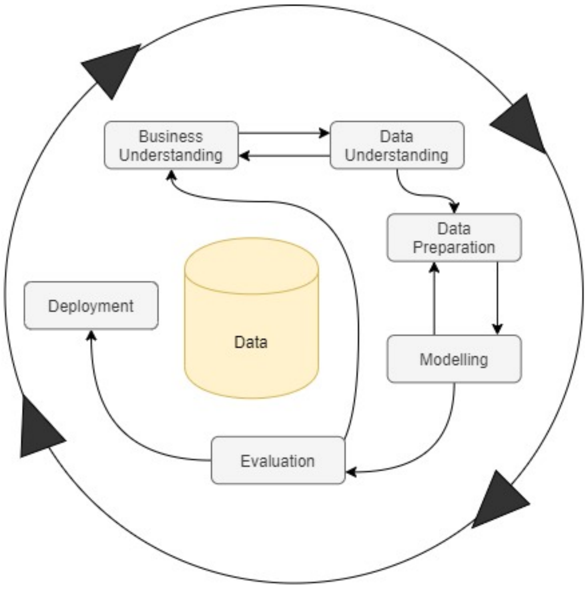


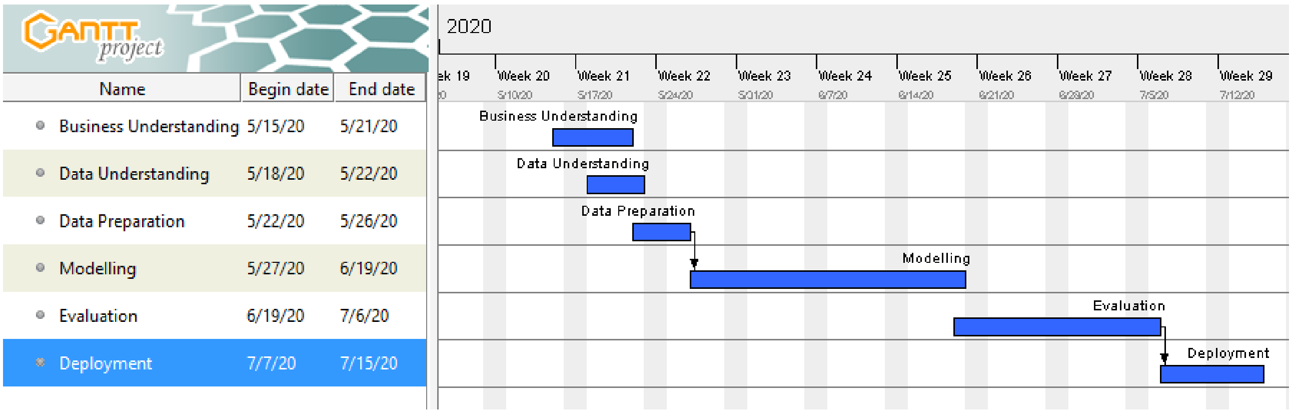
Figure 2.1: CRISP-DM cycle

This document presents a research project development carried as a continuation of my internship at the Informatics Research Unit for Sustainable Engineering (IRUSE) group (to whom we are deeply grateful for providing the Alice Perry building data) at the National University of Ireland Galway (NUIG), therefore the traditional software project management methodology is not applicable.

However, according to Bob Hughes and Mike Cotterell [26] planning in uncertain projects (like research) *is worth doing as long as the resulting plans are seen as provisional.* Hence, this section presents a slightly diffuse but flexible scheduling and planning highly suitable for data science projects, the Cross Industry Standard Process for Data Mining (CRISP-DM) model, which consists of the following phases or stages:

1. **Business Understanding**: where the project objectives are set and understood
2. **Data Understanding**: where the data collection and familiarization takes place
3. **Data Preparation**: where the dataset is preprocessed, cleaned and set for modelling
4. **Modelling**: where models are tuned, calibrated , trained and validated
5. **Evaluation**: where the best models are considered and checked if they satisfy the business objectives and needs
6. **Deployment**: where the final results are delivered, in a wide variety of forms, from a report like this document to an industrial application

Figure 2.1 shows the structure of a CRISP-DM project. This process is iterative, since each time you step forward, revisiting previous phases is usually needed. Figure 2.2 shows an approximate scheduling of the phases of this project where the where the **main workload** for each stage will occur . However, as the CRISP-DM methodology is an iterative approach, for instance, business understanding might be revisited when the evaluation stage is (mainly) taking place. Early tasks, although important, take less time in the diagram because I became familiar with the data during my IRUSE internship.



Name	Begin date	End date
Business Understanding	5/15/20	5/21/20
Data Understanding	5/18/20	5/22/20
Data Preparation	5/22/20	5/26/20
Modelling	5/27/20	6/19/20
Evaluation	6/19/20	7/6/20
Deployment	7/7/20	7/15/20

Figure 2.2: Gantt diagram for the project

2.2.1 Cost estimation

Only one reference on cost estimation for Data Mining was found [27]. However, this paper was rather confusing and had strong assumptions, basing some of its metrics in estimating the cost of misclassifying one observation. That estimation in particular is too difficult for the data this project was carried with, so that approach is not viable.

However, trying to estimate the overall project costs is beneficial in terms of planning is always necessary, at least a time estimation to stay on track. This hour duration estimation was showed in Figure 2.2. Table 2.1 shows an identification of each task and subtask with the estimated hours for each one of them, distributing them so they sum up to the 300 hours planned for a final degree project (according to official documentation).

Task	**Subtask**	**Estimated Duration in hours / percentage**	**Total duration**
Business Understanding	Meeting with Maintainers	5 (1.6%)	25 (8%)
	Literature Review	15 (4.8%)	
	Setting objectives	5 (1.6%)	
Data Understanding	Choosing the datasets	5 (1.6%)	15 (4.8%)
	Reading dataset documentation	5 (1.6%)	
	Data Visualization	5 (1.6%)	
Data Preparation	Choosing variables	10 (3.2%)	25 (8%)
	Cleaning outliers	7 (2.25%)	
	Filling Missing Values	8 (2.58%)	
Modelling	Studying Deep Learning	35 (11.3%)	140 (45.3%)
	Choosing candidate models	35 (11.3%)	
	Learning Framework (Python & Keras)	35 (11.3%)	
	Model implementation	35 (11.3%)	
Evaluation	Testing the models	70 (22.6%)	70 (22.6%)
Deployment	Document writing	35 (11.3%)	35 (11.3%)
Total			310

Table 2.1: Hour estimation for each subtask

However, the level of uncertainty in research projects and my lack of expertise in them make this estimations too unstable. In fact, research projects have many risks involved, since you walk into the unknown. Instead, the estimation duration should be regarded more like a percentage than like an absolute estimation, since risks can increase the duration of the project by unexpected amounts of time.

2.2.2 Risk Identification

In fact, risk identification is much more interesting in this sort of projects than time estimation. Table 2.2 shows the identified risks for the project, their likelihood of happening and their severity.

Task	Potential Risk	Likelihood	Severity
Business Understanding	Not able to meet mantainers	Low	Low
	Lack of literature	Very Low	High
	Objectives unclear	Medium	Critical
Data Understanding	Lack of existing data	Low	Critical
	No documentation on data or hard to understand	Medium	High
Data Preparation	High Data complexity	High	Low
	Inability to clean data	Low	Medium
	Inability to use data	Low	Critical
Modelling	No information on DL	Very Low	Critical
	No models suitable for the problem	Very Low	Critical
	Coding takes longer than expected	High	Medium
	Unable to implement some models	High	Low
Evaluation	Testing was not done properly	High	Medium
	Testing takes longer than expected	High	Medium
Deployment	Not able to delivery on time	Medium	Critical
Levels		Very Low Low Medium High	Low Medium High Critical

Table 2.2: Risk identification

2.2.3 Revisiting Planning

This section reviews if the risks listed in table 2.2 happened and what were the consequences in the planning. Table 2.3 shows the consequences of the identified risks that ended up happening

Task	Potential Risk	Happened?	Consequences
Business Understanding	Not able to meet mantainers	Yes	Online meetings due to the coronavirus outbreak
	Lack of literature	No	
	Objectives unclear	No	
Data Understanding	Lack of existing data	No	
	No documentation on data or hard to understand	Yes	Very few of the public datasets met the requirements
Data Preparation	High Data complexity	No	
	Inability to clean data	Yes/No	Data preprocessing took quite longer than expected being a very challenging task
	Inability to use data	No	
Modelling	No information on DL	Yes/No	Information was available but very deep knowledge was required to meet the objectives and took longer than expected
	No models suitable for the problem	No	
	Coding takes longer than expected	Yes	Understanding Keras to the depth required for this kind of research took longer than expected
	Unable to implement some models	Yes	Not in terms of implementation but the computational resources were not enough to train some of them
Evaluation	Testing was not done properly	Yes	Some trials were done with wrong parameters or different setup than the final one so they had to be discarded
	Testing takes longer than expected	Yes	Models took much longer han expected to train
Deployment	Not able to delivery document on time	No	

Table 2.3: Risk identification revisited

In addition to all the identified risks, the coronavirus outbreak had a big impact in the development of the project, since I had to return from Ireland and developing without physical meetings took longer due to the necessity of scheduling an online meeting each time a decision had to be taken. The estimated impact of all the events listed above was around 30 to 40% more time than the scheduled in table 2.1, specially in the Data Preparation and the Model Testing periods.

3. Deep Learning

3.1 Deep Learning & Artificial Neural Networks

Deep Learning models, are part of the family of ML methods. These methods are widely used in Regression and Classification problems. Although there are many DL approaches, the focus of this project will be Artificial Neural Networks (ANNs).

The first Artificial Neural Network model was proposed in 1943 by Warren S. McCulloch & Walter Pitts [4] as a mathematical model to represent neural/mental activity. Further work was carried by Alexey Ivakhnenko and Lapa [5] in 1967 when they proposed supervised feed-forward multilayer neural network.

However, it wasn't until recent times when Deep Learning got a place in the State of the Art in multiple applications. In 1986, David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams [6] presented the backpropagation algorithm applied to ANN which allowed to train the neurons that make up the network weights based on the error they made when predicting an observation. This work proved that it was possible to train multiple layer perceptrons (MLP) with a fast, simple and no hyperparameter dependant algorithm like backpropagation. However, computational power still fell short. These multiple layer architectures are also referred as Deep Neural Networks or DNNs.

When talking about the breaktrough that caused the comeback of Deep Learning people usually refer to *AlexNet* [7]. In this 2012 project, Alex Krizhevsky, Ilya Sutskever & Geoffrey E. Hinton trained a Convolutional Neural Network (CNN) with a multipleGPU implementation in CUDA. They also included new features such as using the Rectified Linear Unit or ReLU [8] as the activation function instead of the sigmoid function. They used their model for the ImageNet 2012 contest, in which it achieved top-1 and top-5 test set error rates of 37.5 % and 17.0 %5 while the best performance achieved during the previous edition was 47.1% and 28.2% . This motivated more work in the field due to the fact that they used a supervised approach only training with backpropagation.
Nevertheless, in this Final Degree Project the main objective is to work with Time Series. Many

models have been applied to this particular kind of data, but the Recurrent Neural Network (RNN) architecture is the most popular one.

RNNs were popularized by Hopfield in 1982. However, RNNs architectures suffered from the vanishing gradient problem [9](where the gradients calculated for updating the network's weights went towards small values, preventing the network from being further trained). It was in 1997 when Sepp Hochreiter and Jürgen Schmidhuber proposed a novel RNN achitecture, Long Short-Term Memory (LSTM) cells [10]. This architecture solved the problem of vanishing gradients by introducing an internal state into the cells, where important information is stored and kept through time. LSTMs can reset this internal state and update it with relevant new information.
In the following sections, further theoretical explanation on the main three architectures used in this Final Degree Project and their foundations will be carried out.

3.1.1 Perceptron

The basic unit of a neural network is the simple perceptron or neuron.

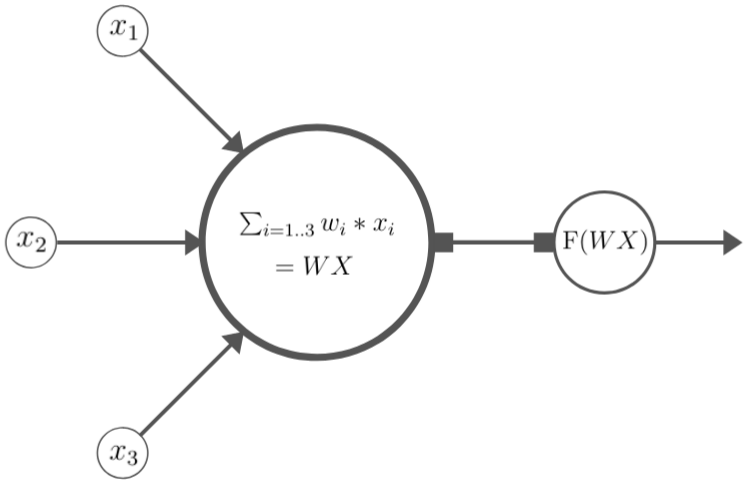


Figure 3.1: Diagram of a Neuron with three inputs which contains a set of weights W, one for each input X and an activation function F for outputing its value

A neuron is a simple unit representing a mathematical function based on the McCulloch-Pitts model [4]. It consists on a set of weights W, one for each input in X. The idea is to make a basic calculation by adding each input multiplied by its corresponding weight and then output a value by applying an activation function to that sum. The original activation function was the binary activation, trying to emulate the biological neurons excitation and information exchange.

$$F(x) = \begin{cases} 1, x \geq 0 \\ 0, x < 0 \end{cases} \tag{3.1}$$

3.1.2 Activation Functions

In this section, the considered Activation Functions (AFs) for the project are explained alongside the reasons to finally using them or not.

Sigmoid

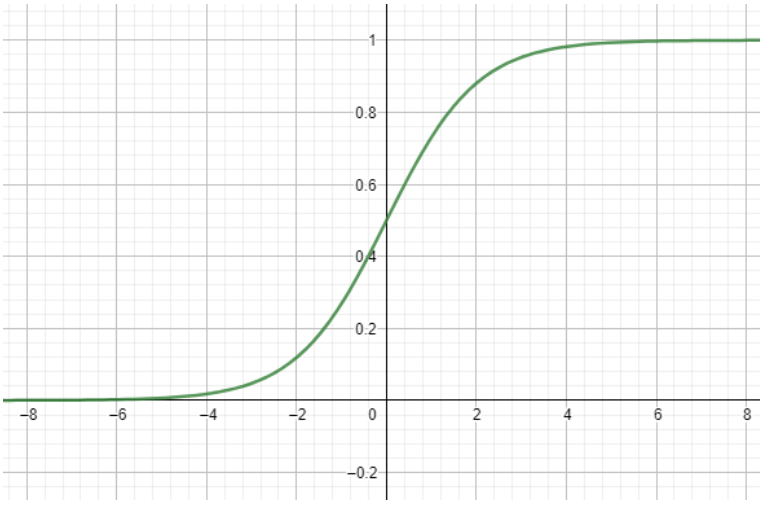


Figure 3.2: Sigmoid activation function

The sigmoid function is a non-linear AF used in neural networks. It's a non linear activation, derivable continuous with a positive derivative in any point. It's defined by the following formula:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3.2}$$

Neal ([11] trough [12]) highlights the main advantages of the sigmoid functions as, being easy to understand and are used mostly in shallow networks.

However, according to Nwankpa [12]: "*the Sigmoid AF suffers major drawbacks which include sharp damp gradients during backpropagation from deeper hidden layers to the input layers, gradient saturation, slow convergence and non-zero centred output thereby causing the gradient updates to propagate in different directions. Other forms of AF including the hyperbolic tangent function was proposed to remedy some of these drawbacks suffered by the Sigmoid AF*".

Despite the fact that this AF used to be one of the most if not the most popular, these drawbacks alongside the existence of new and better approaches make it not suitable for practical use.

Hyperbolic Tangent

Hyperbolic tangent is usually applied over the output of LSTMs, as further sections show (see Figure 3.10), in order to limit their output between 1 and -1. It arose as a solution for the drawbacks for the sigmoid AF.

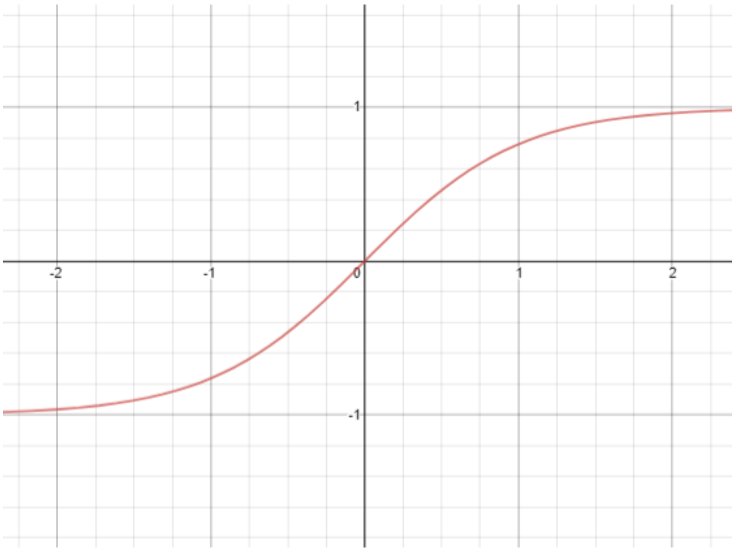


Figure 3.3: Hyperbolic tangent (tanh) function

This is a trigonometric function, and is not usually expressed with another formula, however, it's defined as follows:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.3}$$

Rectified Linear Unit

The Rectified Linear Unit (commonly known as ReLU) was an activation function proposed by Hahnloser in [8]. Dahl et al. [13] showed that usung the ReLU function improves the error rate when using ReLUs instead of sigmoids as activation functions.

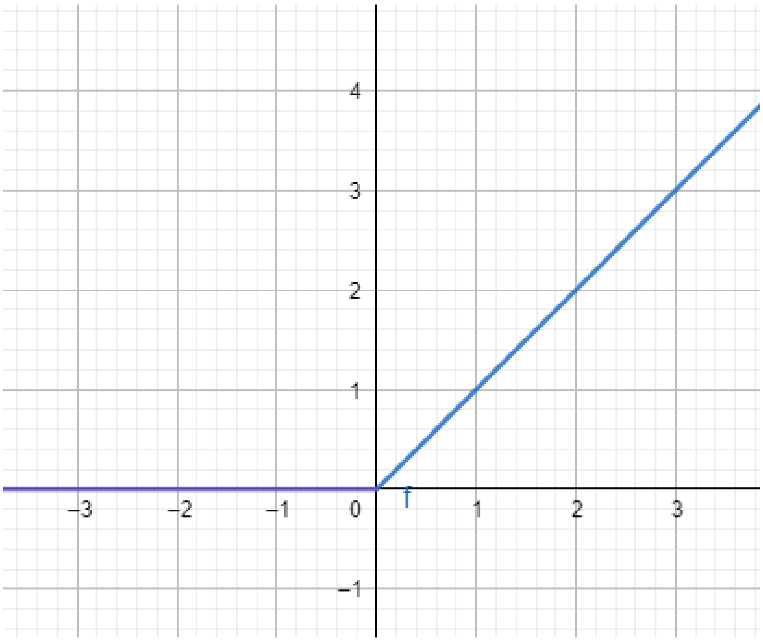


Figure 3.4: Rectified Linear Unit activation function

It's defined as follows:

$$F(x) = \begin{cases} x, x \geq 0 \\ 0, x < 0 \end{cases} \tag{3.4}$$

According to [12], this function rectifies the values of the inputs less than zero thereby forcing them to zero and eliminating the vanishing gradient problem observed in the sigmoid AFs. But the main advantage of ReLU comes also from a computational cost perspective, since it does not make any calculations (i.e. divisions or exponentials) ([14] through [12]).

LeakyReLU

However, ReLUs suffer the problem known as *Dying ReLUs*. The hard 0 activation value in the ReLU makes the gradient 0 whenever the neuron is not activated. According to Maas et al. [15] if the neuron is not initially activated, its weight might not get updated due to the zero gradient, so the weights value get stuck and the network can't improve its predictions. In the same paper,Leaky ReLU was proposed as a way of overcoming this problem. Instead of forcing a 0 value when the input is negative, a small value is returned, controlled by the leaking parameter α. This makes the gradient not 0 in the entire AF domain.

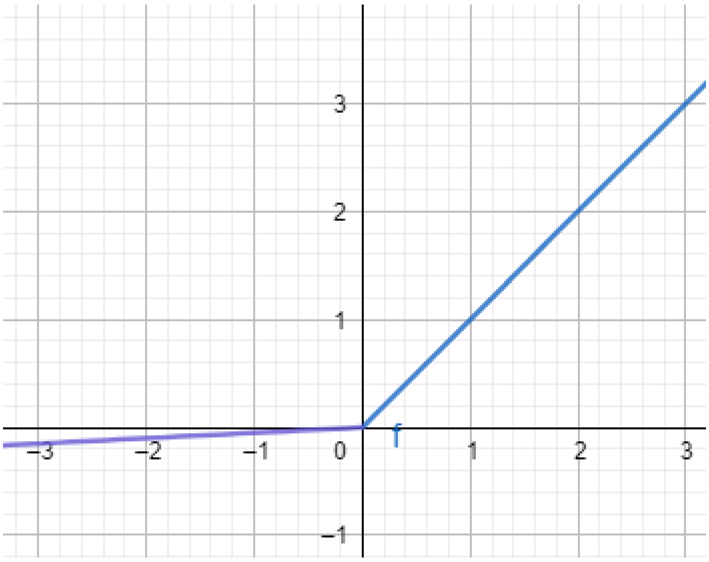


Figure 3.5: Leaky ReLU activation function with $\alpha = 0.03$

It's defined by the following equation:

$$F(x) = \begin{cases} x, x \geq 0 \\ x * \alpha, x < 0 \end{cases} \tag{3.5}$$

3.1.3 Feed Forward Networks

Feed Forward Networks or FFNs are ANNs composed by Dense layers (Figure 3.6) of simple perceptrons or neurons, which propagate the information forward into the next layer (each neuron outputs its value to all neurons in the next layer). We talk about Dense or Fully Connected layers where all neurons are mapped to all neurons in the next layer.
The layers between the first and the output layers are called hidden layers, and they are not always included (this is known as a single layer network) in the ANN.

Although they showed good performance for tabular data, FFNs fell short for other problems, like pattern recognition in handwritten Zip codes. That's why in 1980, Fukushima [16] proposed his Neocognitron as a primitive Convolutional Neural Network, and in 1989, Waibel et al. released their Time-Delay Neural Network (TDNN) used for phoneme recognition, as a way to capture the time-dependant structure of the data.

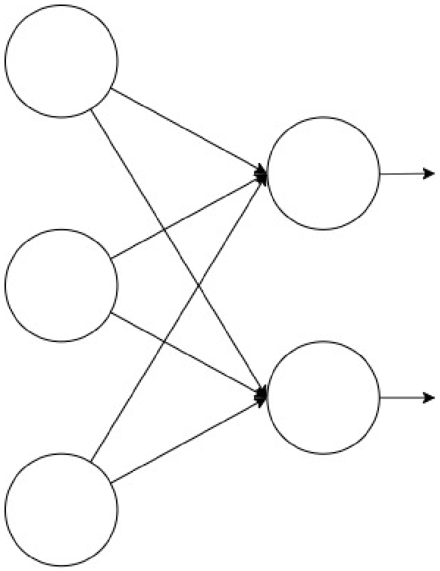


Figure 3.6: Example of a Dense or Fully connected Layer

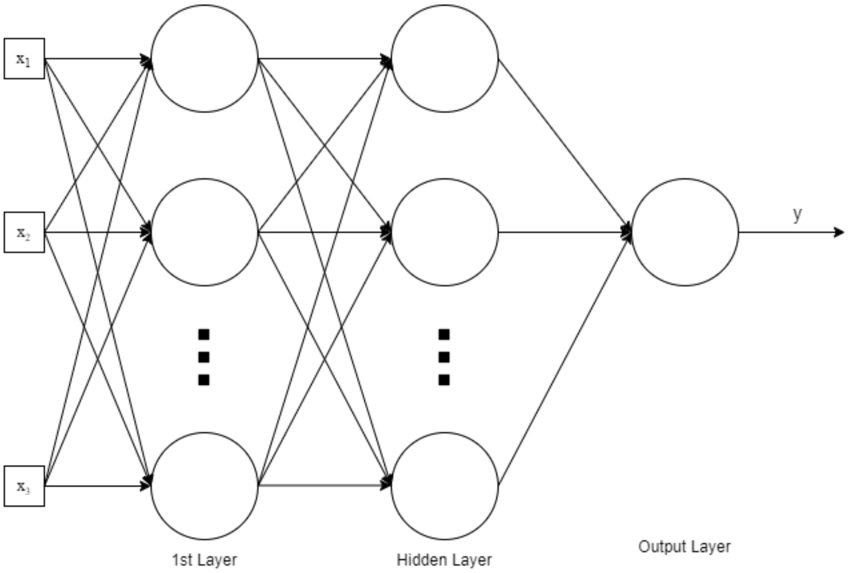


Figure 3.7: Diagram of a 2 Dense Layers DNN

3.1.4 Convolutional Networks

Convolutional Neural Networks or CNNs [17] are special type of Neural Networks, which has shown exemplary performance on several competitions related to Computer Vision and Image Processing. Some of the exciting application areas of CNN include Image Classification and Segmentation, Object Detection, Video Processing, Natural Language Processing, and Speech Recognition.

The motivations behind convolutions in ANNs are many. But the two most important ones,as

LeCun exposed in the LeNet paper [18] considered by many as the first CNN, FFNs present too many parameters when analizing images with hundreds of pixels, and they can't analyse the structure of the data (when flattening an image input, the structure of the original pixels and their distribution gets lost). However, LeCun in the same paper claims that the main deficiency of unstructured nets for image or speech recognition applications is that hey have no built-in invariance with respect to relocations or local distortions of the input (see [18] for further details).

Convolutions

In Convolutional layers the input is a tensor with any dimension (1D,2D,3D...). The idea of convolutions is to apply a filter to extract features from the input data. Since we are working with time series data, 1D convolutions will be applied. Figure 3.8 shows an example of a 1D convolution.

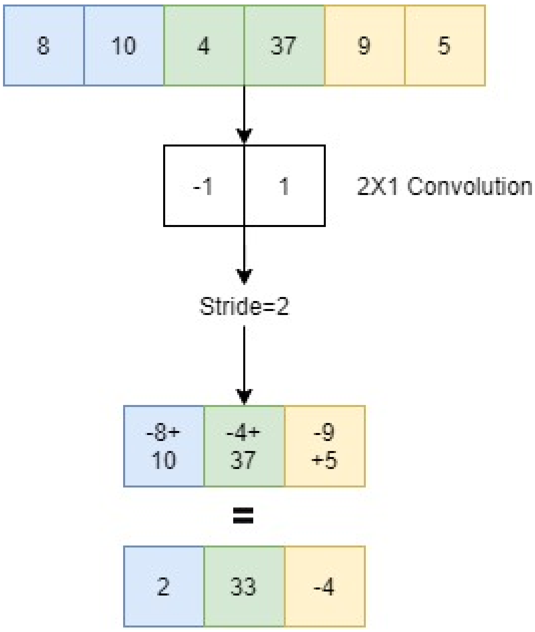


Figure 3.8: 1D-Convolution example over a 1D vector

The values of the convolution (-1 and 1 in the example) are learnt by the network through backpropagation.

Pooling

After applying convolutions and extracting features, pooling layers are applied to make a summary of those features and reduce the dimensionality. These two layers are usually applied one after another, concatenating them and finally feeding the input into a FFN by flattening the last layer. Figure 3.9 shows an example of Max Pooling, the choice for this Final Degree Project.

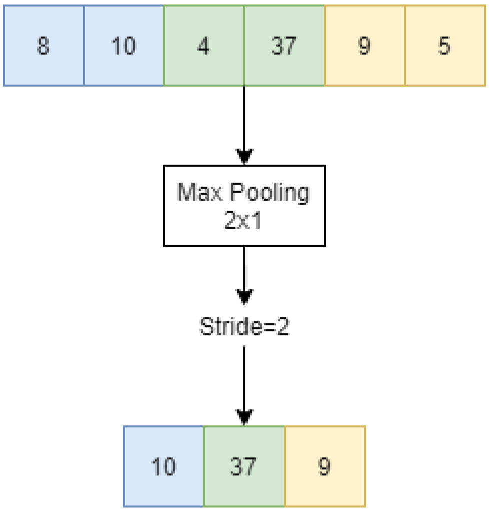


Figure 3.9: Max Pooling example over a 1D vector

3.1.5 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a class of ANN where connections between neurons are not only in a forward direction but also backward, making cycles in the ANN graph. This allows to return information to previous layers for future inputs, allowing for a time-distributed architecture. RNNs started with Rumerlhart's 1986 [6] work, leading to Elman nets [19] y Jordan nets [20]. However, the most popular architectures are the Long Short-Term Memories and the Gate Recurrent Units. In this project, Long Short-Term memories will be the core architecture, due to the time series nature of the energy load data.

Long Short-Term Memory Networks

Long Short-Term memories [10] came up as a solution for the vanishing gradient in neural networks. Their solution was to include a memory cell c and a hidden state h that kept important information throughout the training process and enabling transmission during the sequence chain, making it able to reset it aswell if more relevant information showed up.

To achieve this, the *forget gate* outputs a 1 or a 0 thanks to the sigmoid function σ depending on if the information on the cell state is decided to be kept or not.

The next output is provided by σ in the *update gate* and *tanh* in the *candidate gate*. Using the product of these two values, and then adding it with the product of the *forget gate output* and the previous *cell state*, we get the updated memory cell value.

The last step is to calculate the hidden state, by applying the *tanh* over the cell state and multiplying the resulting value by the output from the *output gate*. Figure 3.10 shows the pipeline for this process. Every time a σ or tanh function is going to be applied as an AF, a different set of weights and bias is applied to the input before the AF.

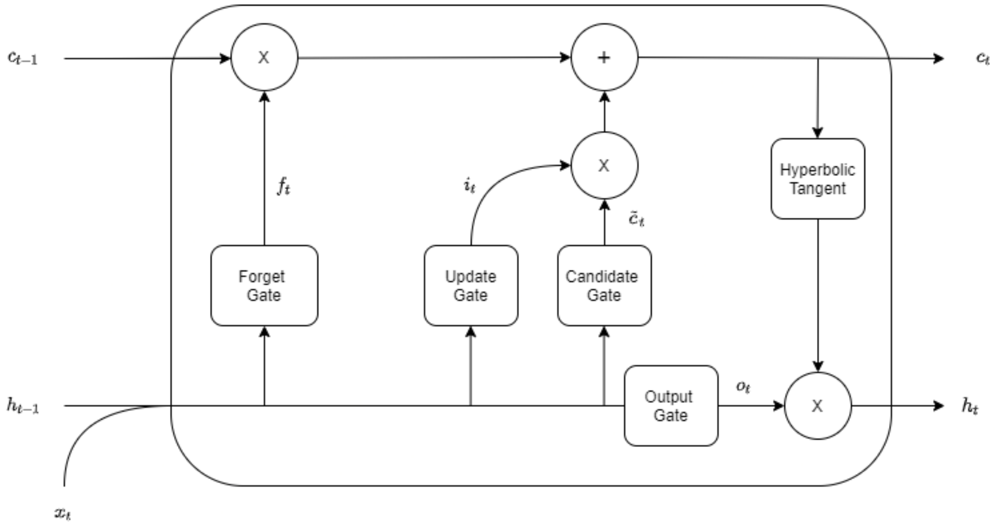


Figure 3.10: LSTM Cell pipeline. Figure adapted from https://en.wikipedia.org/wiki/Long_short-term_memory

The LSTM pipeline can be described by the following equations:

$$f_t = \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right)$$
$$i_t = \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right)$$
$$o_t = \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right)$$
$$\tilde{c}_t = tanh\left(W_c x_t + U_c h_{t-1} + b_c\right)$$
$$c_t = f_t \text{ x } c_{t-1} + i_t \text{ x } \tilde{c}_t$$
$$h_t = o_t \text{ x } tanh\left(c_t\right)$$
$$\hat{y}_t = h_t$$

Figure 3.11 shows an unrolled version of the diagram in figure 3.10. The hidden state is fed into the next into the next timestamp, and the cell state is transferred in a kind of *conveyor belt*, updated only when the forget gate decides that new information is more important to be kept.

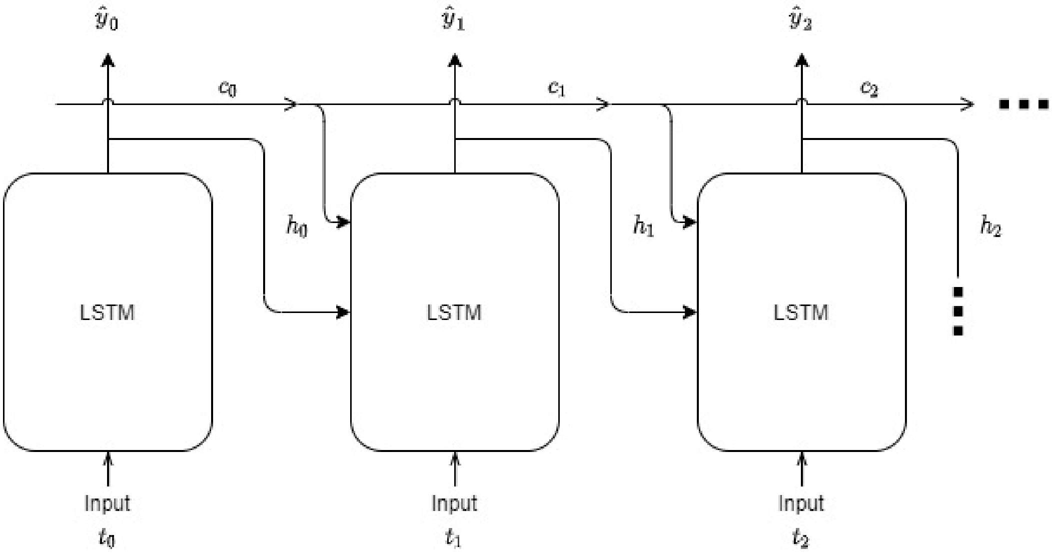


Figure 3.11: LSTM Unrolled Graph. Adaptation from Figure 6.13 from [21]

3.1.6 Other layers

Dropout

Published by Google[22] in 2014, this technique emerged as a solution to palliate over-fitting. The idea is to temporarily cancel some neurons during training ruled by a *dropout* chance. Figure 3.12 shows an ANN with two neurons *dropouted*. They won't be used neither in the forward propagation nor on the backward propagation in that training round. However, dropout is not permanent, each time different neurons are blocked. This makes the network not rely on specific neurons, thus reducing overfitting due to heavily influencing characteristics

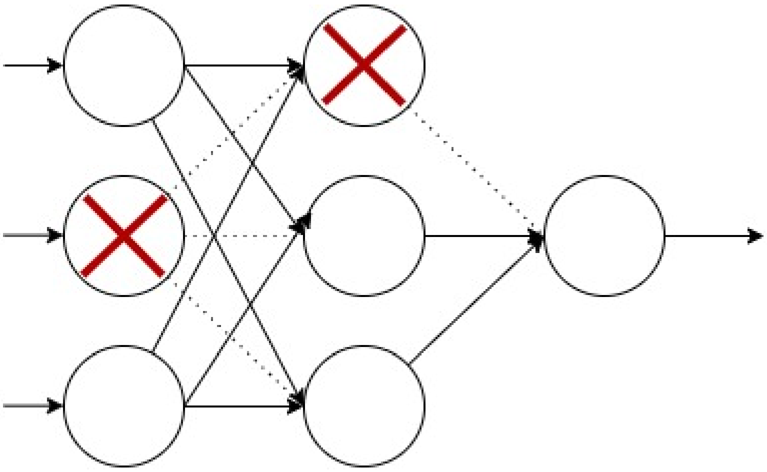


Figure 3.12: Dropout Example

Concatenation

Sometimes, two models are merged to create a Hybrid model. In order to merge models, a concatenate layer is used. In this layer, both model outputs are merged in order to be fed to another layer.

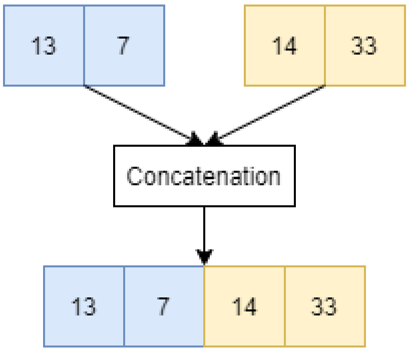


Figure 3.13: Concatenate Layer example

Flattening

Flattening layers are used after a convolutional block, so the convolutions+pooling output can be fed into a dense layer. Figure 3.14 shows a diagram of this layer's behaviour.

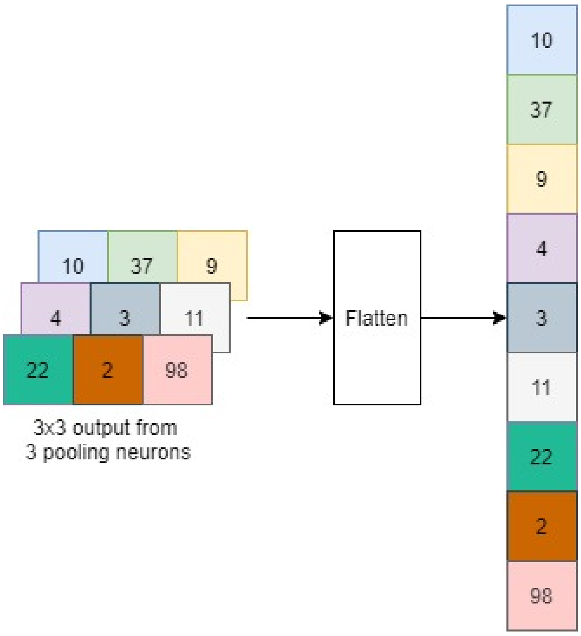


Figure 3.14: Flatten Layer example

3.1.7 Training process

Once the topology of the network is set, the training process begins. The training and test splits from the data available must be hermetic so the algorithm does not see the test data at any stage of the training process.

Epochs

An epoch is a whole pass (forward and backward) over all the training sample. Multiple epochs are necessary in the training process due to the iterative nature of gradient descend based methods like backpropagation.

Mini-Batches

Training set is usually divided in subsamples called mini-batches. The idea is to calculate the gradient for small portions of the sample instead of the entire sample. This allows to take more precise steps when approaching the minimum of the cost function. However, determining the mini-batch size is not trivial. If the minibatch is too small, the training process will take too long, and if it's too big the descend calculation might not be optimal. Usually batch size is set as powers of two due to the binary nature of memory management in computers.

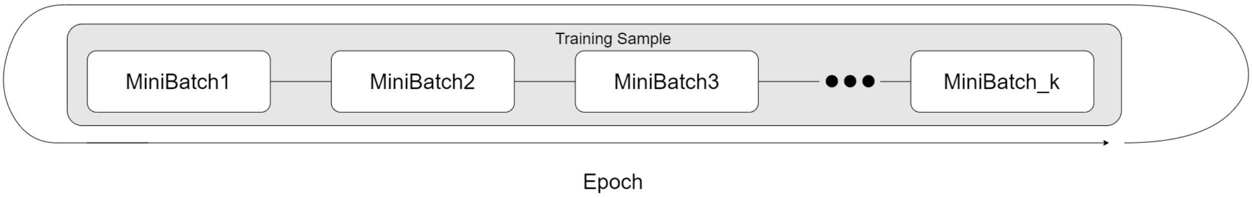


Figure 3.15: Train sample division in batches and epoch representation

Keras considerations: There is a key parameter for training networks for sequence prediction, the `shuffle` parameter, set to `True` by default. This will create batches automatically by shuffling the samples. However, when training for sequence data, it is better to set this parameter to `False`. This will group observations in batches sequentially, which leads to more representative calculations of the gradient per batch.

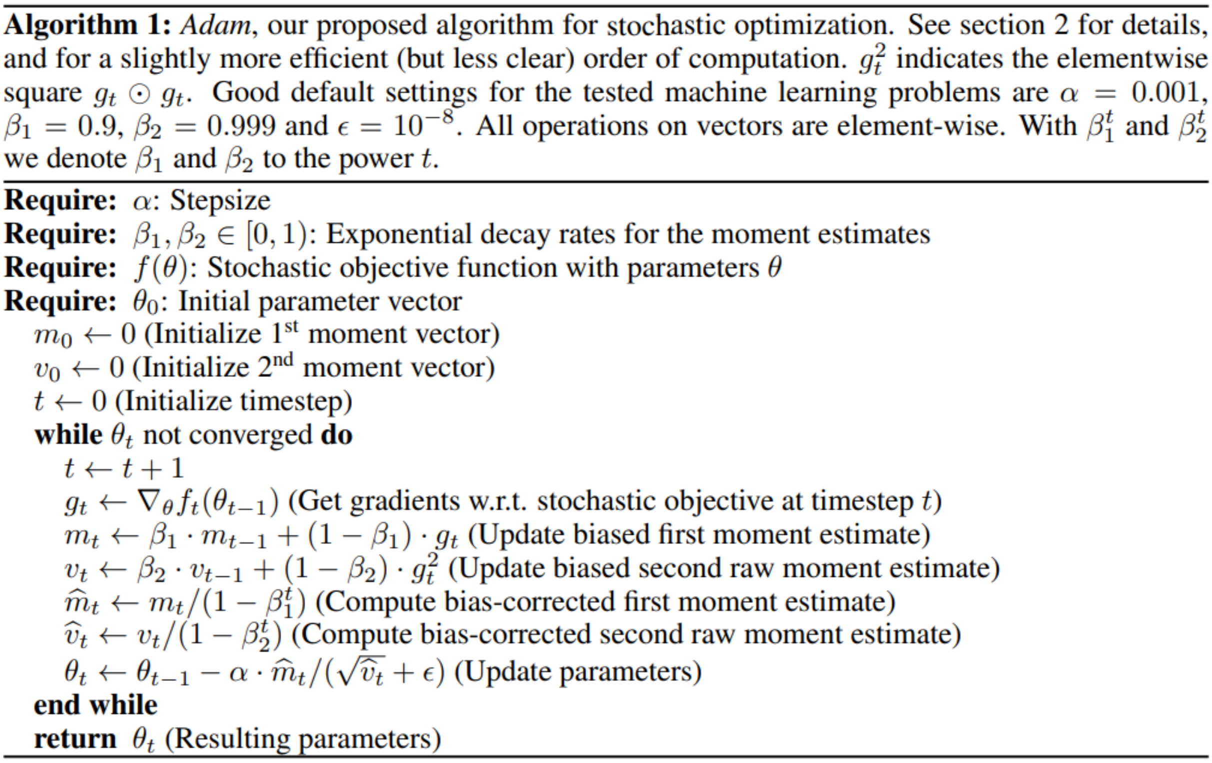
Loss Function

The loss function tells the neural network about how well (or how bad) its predictions are doing. It takes as inputs the real response values and the predicted response values and computes values based on the difference between these two arguments.
The direction of descent is calculated through the derivative of this function. Since we are handling continuous data, in this project the choice was Mean Square Error Loss (also known as Quadratic loss or L^2 loss), defined as follows:

$$\mathcal{L}_{MSE}(y, \hat{y}) = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{3.6}$$

Optimizer

The function of the optimizer is to calculate the magnitude of the update values for the weights in the ANN. Many optimizers exist, although the adaptive gradient methods seem to be the most popular ones [23]. Adam [24] came as a solution to unify AdaGrad and RMSprop. Its strengths rely on using the history of gradient values to adapt the learning rate and the magnitude of the update value for the weights using moving averages and squared gradients. In Figure 3.16 the algorithm extracted from the original paper is showed.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 3.16: Original adam algorithm (extracted from [24])

3.1.8 Training Algorithms

When training DNNs, there are many alternatives depending on what the training environment is (data driven, simulation environments, etc). In this project, backpropagation algorithms will be used.

Backpropagation

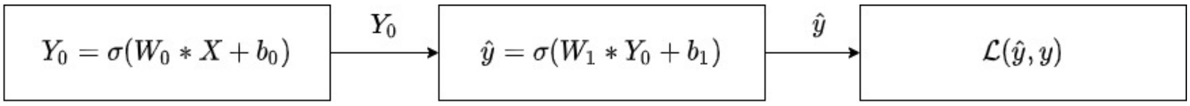
Backpropagation [6] algorithm is used when training neural networks based on data. The idea behind the algorithm is the following iterative process:

1. Present a training input and propagate it to get an output based on the weights (forward propagation).

2. Compare predicted value and real value to calculate an error.
3. Calculate the gradient of the error with respect to the network weights .
4. Update the weights based on the gradient (backward propagation)

However, the problem of very deep neural networks is that the concatenation of derivatives of successive layers results in very small gradients, which can result into the **vanishing gradient** problem, where the network gets stuck and can't update its weights. This problem is (in fact it was so common that it motivated the LSTM development) present in RNNs too.
The vanishing gradient comes from the fact that weights are updated through the gradient, this is, the derivative of the loss function with respect to the weights. In order to backpropagate this error, gradients are concatenated through the rule chain, thus, the first layers of a DNN depend on the gradient of the last values.

Figure 3.17 shows the computational graph of a two layer FFN, which is just a simplified mathematical representation concatenating outputs. Underneath the chain rule for the backprop algorithm is showed to update the first layer weights.


$Y_0 = \sigma(W_0 * X + b_0)$
Y_0
$\hat{y} = \sigma(W_1 * Y_0 + b_1)$
$\hat{y}$
$\mathcal{L}(\hat{y}, y)$

Chain Rule for updating W_0

$$\frac{\partial \mathcal{L}}{\partial W_0} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Y_0} \frac{\partial Y_0}{\partial W_0}$$

Figure 3.17: Computational Graph for a two layer FFN

As it can be seen, the derivative of the sigmoid is present twice, when calculating the partial derivative of $\hat{y}$ and Y_0. What does this imply? When a DNN is very deep, many sigmoid derivatives will be multiplied sequentially. Figure 3.18 shows the values this derivative, $\sigma\prime(x)$, takes:

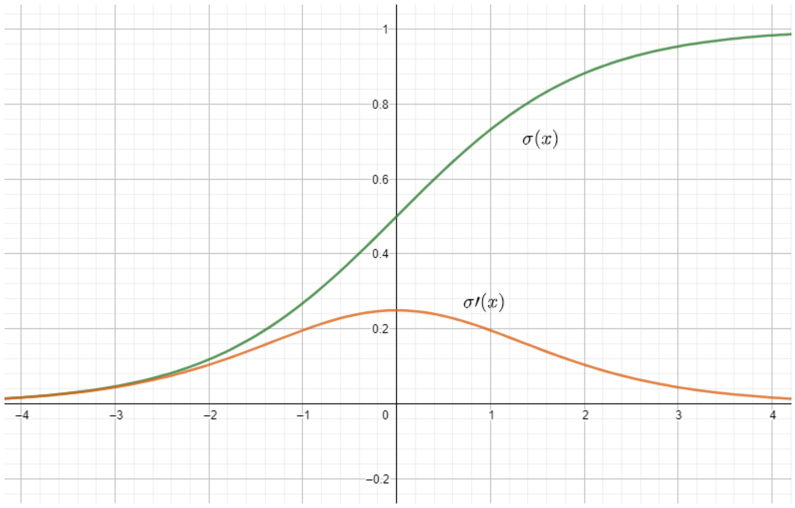


Figure 3.18: Sigmoid function and its derivative

$\sigma\prime(x)$ takes small values, and non zero values only when the input is close to zero. This and the chain rule are the causes of the vanishing gradient phenomenon. If the network is very deep, many small values will be multiplied, as Figure 3.19, therefore the first layers weight updating gets stuck due to a very small gradient value.

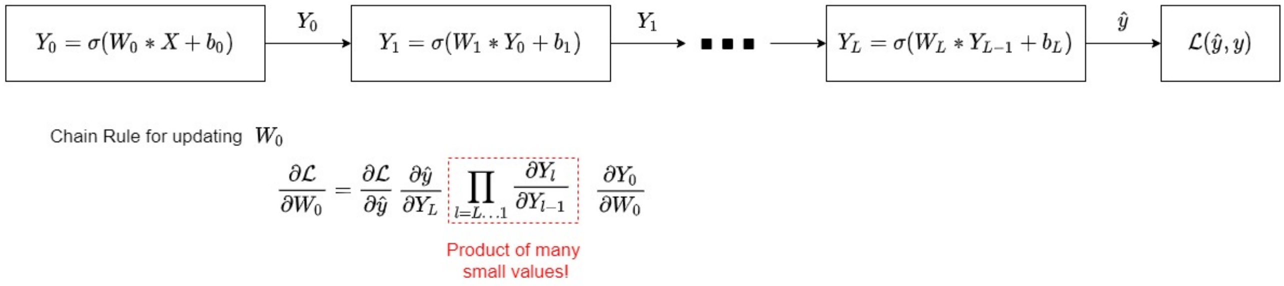

$Y_0 = \sigma(W_0 * X + b_0)$
Y_0
$Y_1 = \sigma(W_1 * Y_0 + b_1)$
Y_1
■ ■ ■
$Y_L = \sigma(W_L * Y_{L-1} + b_L)$
$\hat{y}$
$\mathcal{L}(\hat{y}, y)$
Chain Rule for updating W_0
$\frac{\partial \mathcal{L}}{\partial W_0} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Y_L} \prod_{l=L...1} \frac{\partial Y_l}{\partial Y_{l-1}} \frac{\partial Y_0}{\partial W_0}$
Product of many small values!

Figure 3.19: Computational graph for a Deep neural network

This can also be caused by small weights initialization. The same happens the other way round with other activation functions or when weights take very large values ; big gradients will cause **exploding gradients**

Backpropagation Through Time

Backpropagation Through Time, or BPTT, is the Backpropagation version for RNNs. BPTT works by unrolling the RNN. Each time consists of one copy of the network which gets fed the previous timestep information alongside new information. Errors are calculated and accumulated for each time step. The network is rolled back up and the weights are updated.

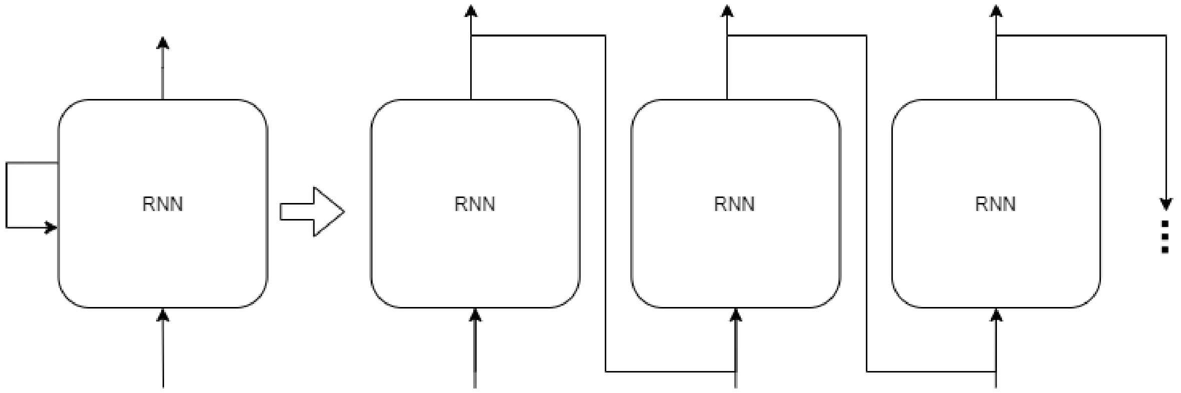


Figure 3.20: Unrolling graph

The iterative process goes as follows

1. Get the sequence of input values and the corresponding sequence of output values
2. Unroll the network and calculate errors in each timestep, accumulate them and calculate gradients.
3. Roll up the network
4. Update weights

However, this approach has its drawbacks. The longer the sequence, the higher the computational cost. This also can cause vanishing/exploding gradients, since a long unrolling is like having a neural network with many layers, thus calculating many derivatives falls into the same faults as the very deep neural networks.

Truncated Backpropagation Through time

Truncated Backpropagation Through Time, or TBPTT, addreses the problem of too long sequences in BPTT. The idea is to process the sequence by lots, calculating the gradient for subsequences of the input sequence instead of doing it as a whole. It's ruled by two parameters, k_1 for the input sequence size and k_2 for the accumulated steps for BPTT. It goes as follows:

1. Get the sequence of k_1 input values and the corresponding sequence of output values
2. Unroll the network and calculate errors in each timestep, accumulating them for k_2 steps
3. Roll up the network and calculate gradient of the error
4. Update weights

All $k_1, k_2 \in \mathbb{N}$ value combinations restricted to $k_1, k_2 < n$ are possible. However, Keras implementation forces $k_1 = k_2 = k$, where k is equal to the choice of window size for previous values to use as predictors.

3.1.9 Validation methodology

In order to test model performance, the following metrics will be used:

Error measurements for one-step ahead prediction

RMSE:

$$\sqrt{\frac{1}{N}\sum_{i=0}^{N}(y_t - \hat{y}_t)^2} \tag{3.7}$$

CVRMSE:

$$\frac{1}{\bar{y}}\sqrt{\frac{1}{N}\sum_{i=0}^{N}(y_t - \hat{y}_t)^2} \tag{3.8}$$

CVRMSE provides an indication on *how much does the errors variate with respect to the mean* over one (if multiplied by 100, the percentage of deviation would be obtained). This allows for comparison between datasets, since dividing by the mean *gets the scale of the data out of the statistic value.*

Error measurements for sequence prediction

Time-RMSE (as proposed by [25]):

$$\sqrt{\frac{1}{N}\sum_{t=0}^{N}\frac{1}{scope}\sum_{i=0}^{scope}(y_t[i] - \hat{y}_t[i])^2} \tag{3.9}$$

Time-CV(RMSE) (as a proposal for CV(RMSE) adaptation):

$$\sqrt{\frac{1}{N}\sum_{t=0}^{N}\frac{1}{scope}\sum_{i=0}^{scope}\frac{(y_t[i] - \hat{y}_t[i])^2}{\bar{y}_t^2}} \tag{3.10}$$

4. Case study

4.1 Available Datasets

According to Wang et al., due to the private nature of Energy Load data, many companies are hesitant to release their data [1]. In the same paper, they showcased a table with Open Load Datasets. Table 4.1 shows the information they gathered about the most popular open energy load datasets

Name	**Brief Description**	**Frequency**	**Duration**
Custom Behavior Trials	Smart meter read data Pre- and post-trial survey data	Every 30 min	2009/9-2011/1
Low Carbon London	Smart meter read data Electricity price data Appliance and attitude survey data	Every 30 min	2013/1-2013/12
PecanStreet	Residential electricity consumption data Electric vehicle charging data	Every 1 min	2005/5-2017/5
Building Data Genome	Non-residential building smart meter data Area,weather and primary use type data	Every 1 hour	2014/12-2015/11
UMass Smart	Residential electricity consumption data	Every 1 min	One day
Ausgrid Residents	General consumption data Controlled load consumption data PV output data	Every 30 min	2010/7-2013/6
Ausgrid Substation	Substation metering data	Every 15 min	2005/5-
GEFCom 2012	Zonal load data Temperature data	Hourly	2003/1-2008/6
GEFCom 2014	Zonal load data 25 weather station data	Hourly	2005/1-2010/9
ISO New England	System load data Temperature data Locational marginal pricing data	Hourly	2003/1-
NUIG Alice Perry	Air Handling Units load data Weather Station data	Hourly	2018/5-2020/2
Tucson Electrical Power	Tucson (Texas) Area Load Data Tucson Airport Weather station	Hourly	2015/07-2018/07

Table 4.1: Open Energy Load datasets from the Wang et al. [1] review

For this project, in table 4.1 a basic description on how they used the datasets and two key points that were needed for this project; the existance of weather data in the dataset and if it was a residential building or not.

Name	**Type of forecast in references**	**Weather Data?**	**Residential Building?**
Custom Behavior Trials	Short term	No	Yes
Low Carbon London	Short term	No	Yes
PecanStreet	No references	No	Yes
Building Data Genome	No references	Yes	No
UMass Smart	Ultra short term	No*	Yes
Ausgrid Residents	No references	No	Yes
Ausgrid Substation	No references	No	Not a Building
GEFCom 2012	Short & Medium Term	Yes	US Utility with 20 zones
GEFCom 2014	Short & Medium Term	Yes	AUS Utility with 10 zones
ISO New England	Short Term	Yes	Not a Building (Zones)
NUIG Alice Perry	Short Term	Yes	No
Tucson Electrical Power	Short Term	Yes	Not a Building (Zones)

*There is a new version of the dataset

Table 4.2: Information available in each dataset and type of source

4.2 Techniques review

Name	Techniques used in references
Custom Behavior Trials	Thefting detection with customer clustering & SVM [28]
	Pattern extraction and customer classification with K-SVD and SVM [29]
	Deep Learning household load prediction using pools of neighbours [30]
	Clustering customer behaviour, reducing dimensionality with Markov models [31]
	Load forecast with a non-parametric approach [32]
	Load forecast with Clustering+MLP [33]
	Load forecast with Boosted Quantile Regression [34]
	Behavior indicators through a cross-domain feature selection and coding approach [35]
	Load profiling with Clustering [36]
	Household characteristic extraction with Unsupervised learning and regression [37]
	Household Load data compression and reconstruction [38]
	Socio-demographic Information Identification from Load data with Convolutional Networks [39]
Low Carbon London	Costumer clustering with C-Vine Copula Mixture Model [40]
	Peak Load Estimation with clustering and quantile-based probabilistic approach [41]
PecanStreet	X
Building Data Genome	X
UMass Smart*	Power Consumption Profiling with Time-Frequency (TF) based clustering [42]
	Layered clustering for load profiling (local clusters into global) [43]
Ausgrid Residents	X
Ausgrid Substation	X
GEFCom 2012	Load forecast with ARX,Echo State Networks and Wavelets while tuning with genetic algorithms [44]
	Load Forecast via quantile regression averaging on sister models [45]
	Load forecast with parametric models [46]
	Load forecast in two stages using nonlinear Lasso and Splines [47]
	Hierarchical load forecasting with Gradient boosting machines and Gaussian processes [48]
	Electric load forecasting and backcasting with Kernel regression, random forests and splines [49]
	Load forecast with Gradient Boosting based splines [50]
	Weather station selection with GEFCom benchmark model [51]
GEFCom 2014	Probabilistic load forecasting using kernel density estimation (KDE) and quantile regression [52]
	Probabilistic load forecasting semi-parametric regression, simulation and quantiles [53]
	Probabilistic load forecasting with forecast combination and residual simulation [54]
	Probabilistic load forecasting with a hybrid model of kernel density estimation and quantile regression [55]
	Probabilistic load forecasting with non parametric Nadaraya–Watson estimators [56]
	Probabilistic load forecasting with Lasso [57]
	Probabilistic load forecasting with Generalized additive models and Quantile regression [58]
	Load forecasting with Tao's benchmark model** incorporating recency effect through moving averages [59]
ISO New England	Paper cited [59][60] [61] but none of those papers used the dataset.
North Carolina Electric Membership Corporation (NCEMC)	Incorporating Rel.Humidity info into Tao's benchmark model [60]
	Incorporating Wind info into Tao's benchmark model [62]
	Temperature scenario generating with bootstraping,date shifting, fourier transformations among ohters [61]
NUIG Alice Perry (private dataset)	Load prediction with LSTM and seasonal clustering [63]
Tucson Electrical Power	LSTM for short term load forecast [64]

**There is a new version of the dataset*

*** Tao's benchmark model was used as basic model for the GEFCom 2014, a vanilla linear model*

Table 4.3: Techniques review

Table 4.2 represents a paper review in which a short summary on what techniques were used in each paper from [1] and dataset and for what purpose is shown. As it can be seen,at the time of these papers, Deep Learning is still emerging in the Load Forecast field and there are not many approaches using these models.

However, later work from authors like Marino et al.[65], Wilms et al.[66] or Gasparin et al. [25] shows that DNNs, specifically RNNs have remarkable performance in the Load Forecast field, even outperforming some of the methods listed in Table 4.2. In this project DNNs architectures will be presented an tested on a real world problem, the NUIG Alice Perry dataset.

4.3 NUIG Alice Perry dataset exploration

In this dataset there are 824338 observations in minutely format corresponding to the aggregated load of the NUIG's Alice Perry building Air Handling Units (AHUs) heating energy load. A weather station in the same building provides 16731 observations in hourly data for ten variables ('Batt [V]', 'PTemp [oC]', 'AirTemp [oC]', 'RH [%]', 'Slr [kW/m2]', 'Slr [kJ/m2] Tot', 'WindSpeed [m/s]','WindDir [deg]', 'Gust 3s Avg [m/s]', 'BP [mBar]'). A group of sparse observations corresponding to a 2016 day were discarded.

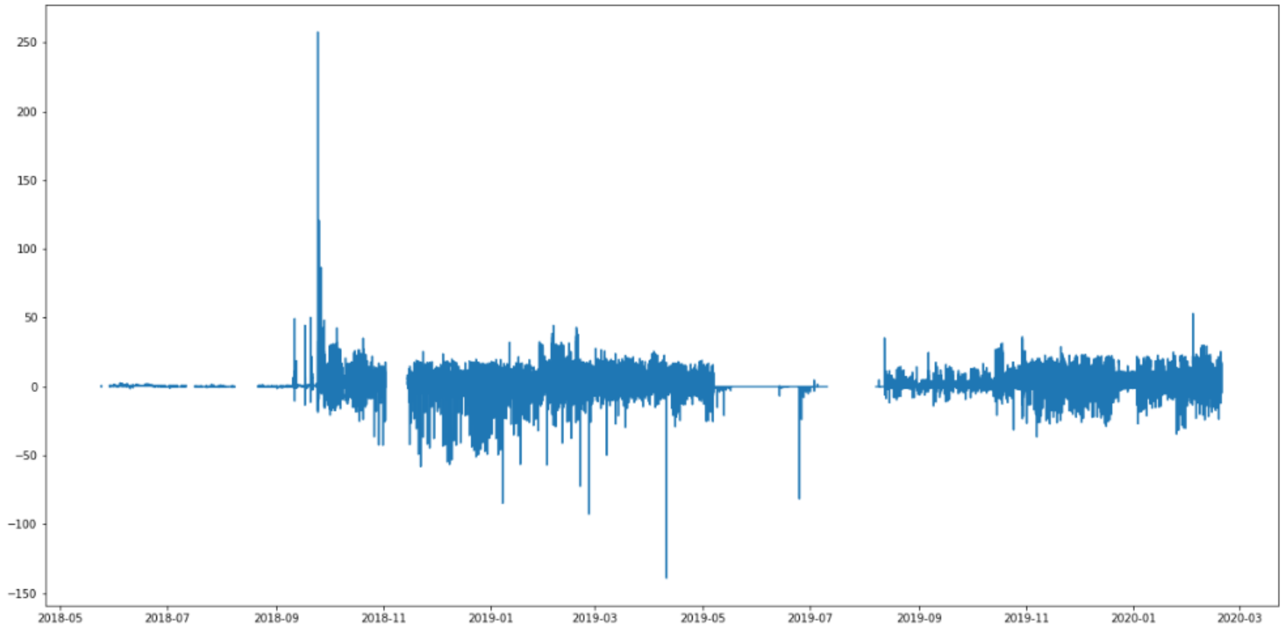


Figure 4.1: Load data in the original scale (minutely data). Gaps represent missing values

The AHUs load data, in kWh, is aggregated into hourly data using the mean. Afterwards, it is merged (outer merge) with the weather station data, making the final dataset with 15270 hourly observations, which presents missing values or NAs in the distribution indicated by Table 4.3. Figure 4.2 shows a time series plot portraying the location of the NAs. Figure 4.3 shows the time series for weather variables.

Varaible	**Mean**	σ	**Minimum**	**Maximum**
Timestamp			2018-05-24 15:00:00	2020-02-19 20:00:00
Load	4.111483	4.373992	-4.65141	22.7941
AirTemp [oC]	9.973651	4.880410	-3.1177	29.4605
BP [mBar]	1010.784819	12.828838	946.091	1045
Batt [V]	13.538727	0.392777	8.82636	13.9718
Gust 3s Avg [m/s]	4.267261	2.268860	0	16.8267
PTemp [oC]	14.522938	5.307262	0.785	37.2435
RH [%]	83.855687	12.643092	26.2922	100
Slr [kJ/m2] Tot	5.752758	9.602083	0	54.2645
Slr [kW/m2]	0.095862	0.160034	0	0.9065
WindDir [deg]	199.253634	89.598314	0	350.942
WindSpeed [m/s]	2.651774	1.435264	0	10.1618

Table 4.4: Statistics for variables in the hourly dataframe

Varaible	**Number of NAs**
Timestamp	0
Load	1500
Record	789
Batt [V]	811
PTemp [oC]	811
AirTemp [oC]	811
RH [%]	811
Slr [kW/m2]	811
Slr [kJ/m2] Tot	811
WindSpeed [m/s]	811
WindDir [deg]	811
Gust 3s Avg [m/s]	811
BP [mBar]	811

Table 4.5: NAs by variable after merging datasets and aggregating by hourr

As it can be seen in Figure 4.3, there are negative values for load. According to data providers, the load is estimated with the following formula:

$$\Delta T = \text{Maximum Airflow } * \text{Valve }\% \tag{4.1}$$

Where $\Delta T = T_{flow} - T_{return}$. Due to Load being estimated, then it maybe the case that $T_{flow} - T_{return}$ inverts when the heating valve is closed so what should be warmer/colder is the other way round (and this depends on the design of the hydronics network).

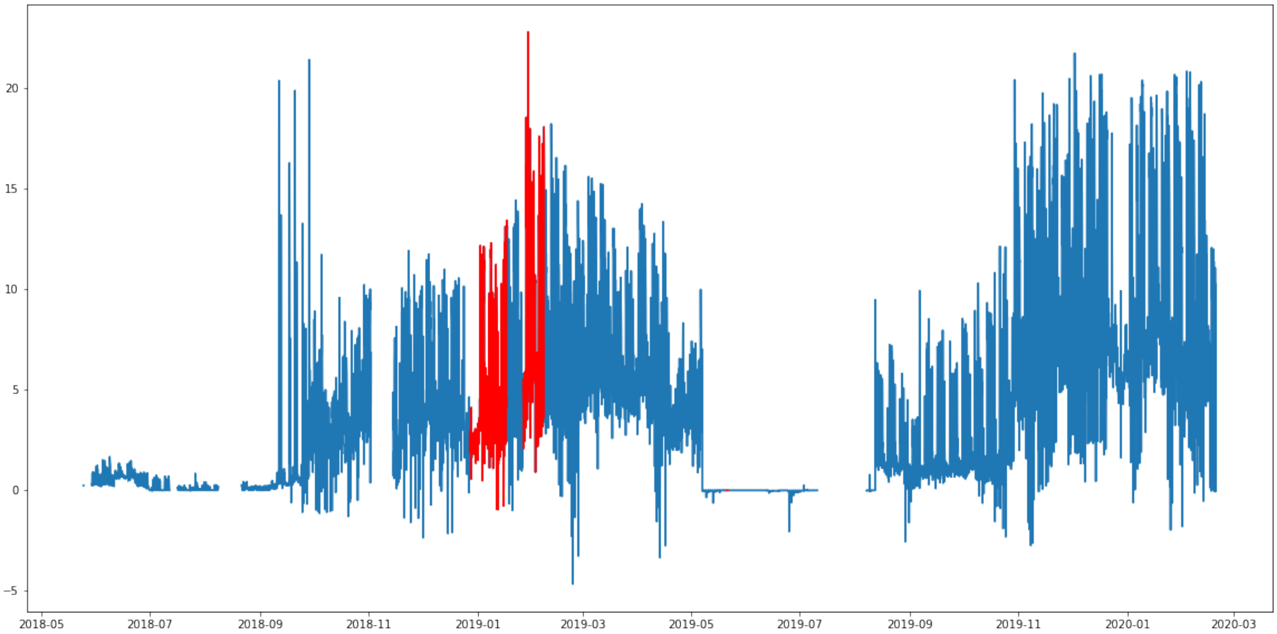


Figure 4.2: Full Hourly data series for load data. Blank spaces represent NAs in load data, and red values represent missing values in the weather data

4.3.1 Cleaning the dataset

As seen in Figure 4.2, between May and September both in 2018 and 2019 the time series takes very small values. These months are the warmer ones, and the time series represent energy consumption from the AHUs in heating mode. Therefore it doesn't make sense to study the heating load in these months, thus they will be removed.

There are also some dates that will be removed. These weeks correspond to the periods the Alice Perry building is closed during winter holidays. Therefore, removing this weeks with a different behaviour should improve the overall performance. Figure 4.4 shows, in purple, the removed periods from the dataset. Figure 4.5 shows the final data with these periods removed.

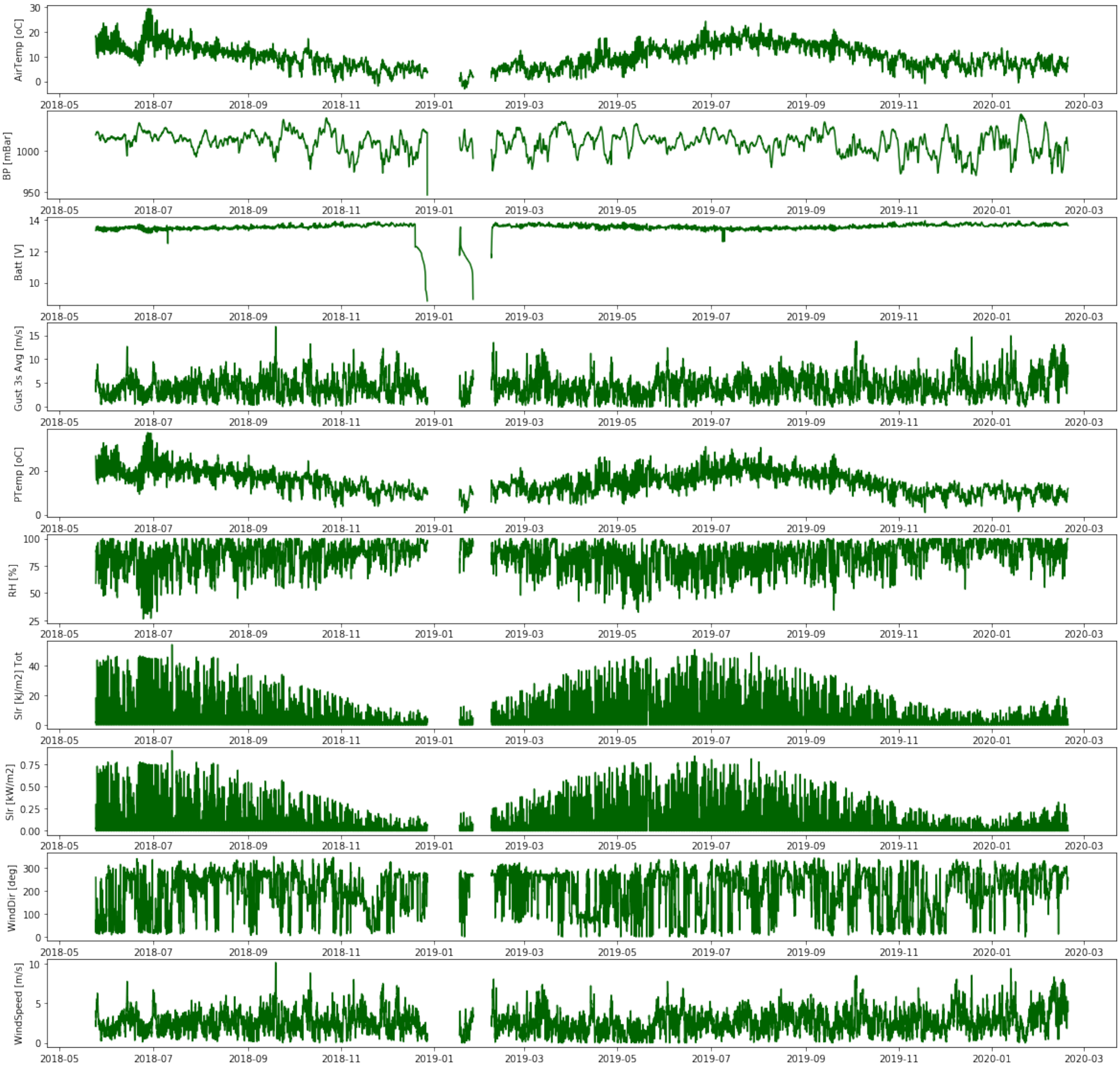


Figure 4.3: Full Hourly data series for weather data. Blank spaces represent NAs in the series

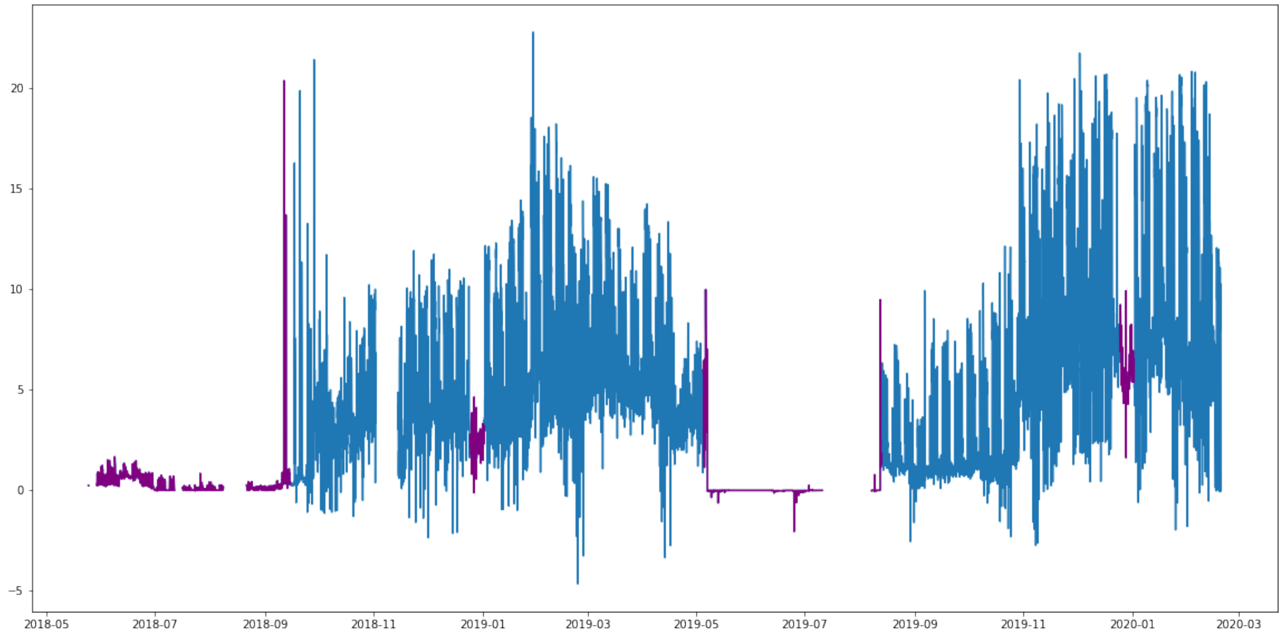


Figure 4.4: Full Hourly data series for load data. Purple shows the dates that will be dismissed due to their different behaviour from what the main series represents

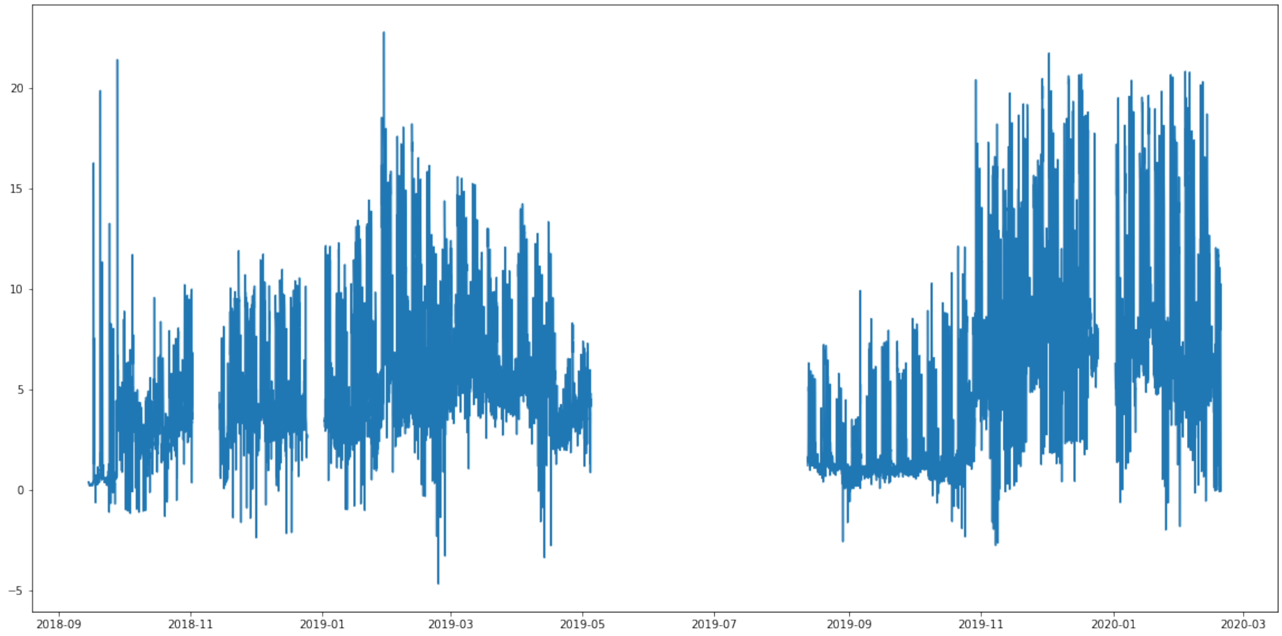


Figure 4.5: Final full Hourly data series for load data after preprocessing

4.3.2 Weather NAs imputation

In order to fill the gaps in Figure 4.3, information from a nearby weather station placed in Atherny, 25km to the east from Galway city. Data is provided by Met Éireann and available at their official website.

However, this dataset does not contain all the variables the NUIG weather station has. Table 4.6 shows the correspondence between both datasets and the conversion needed in order to fill the NAs in the NUIG dataset. These variables will be used as predictors for the Energy Load forecast, and the rest of the available variables will be discarded.

Variable	**NUIG Name**	**Met Éireann Name**	**Conversion/Observations**
Barometric Pressure	BP [mBar]	msl	*Met.ie variable is measured at sea level. NUIG measures at 21m above the sea level, so the difference in pressure of 21m of height must be added before filling*
Relative Humidity	RH [%]	rhum	*No conversion needed*
Wind Speed	WindSpeed [m/s]	wdsp	*Met.ie provides windspeed in knots (kt), conversion to m/s is needed (1knot=0.5144m/s)*
Air Temperature	AirTemp [oC]	temp	*No conversion needed*
Solar Radiation	Slr [kW/m2]	Not Available	*This variable is not provided by Met.ie. However, authors supsect it's key to the problem, therefore it will be included as a predictor*

Table 4.6: Table of correspondence between NUIG dataset and Met.ie dataset

In order to impute the NAs for the Solar Radiation variable, regression using the other variables, SARIMA models and using Load mean values for Day-Hour combinations were tested, but none of them was successful. Therefore, filling the gap with the data from the same period but from 2020 was the way to go, which gives a noisy but similar pattern to what the 2019 period should've been

As Figure 4.3 shows, there is a sudden drop before the NA gap (on the 5206 observation, corresponding to the date of 13:00 @ 2018-12-27), and it is believed to be caused by a problem with the sensor. Therefore, this value will also be treated as a missing value and imputed with the weather dataset. Figure 4.6 shows the weather variables with their imputed values coloured in maroon.

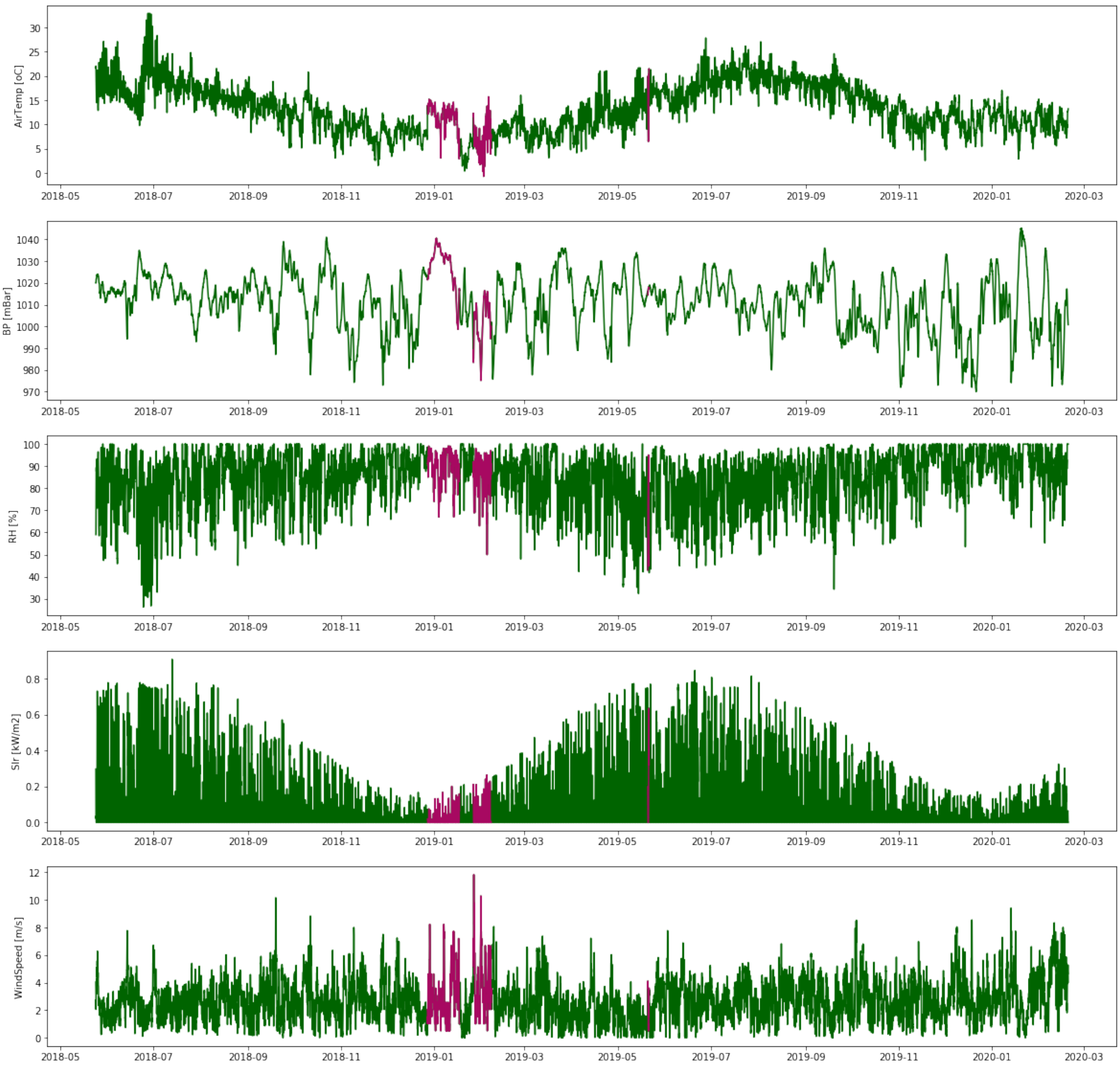


Figure 4.6: Weather data after imputation. Maroon colour indicates the filled values

4.3.3 Time codification

In order to use the categorical data ($Hour_t$ and Day_t) to predict $Load_t$, we need to transform these variables. We can't use the $Hour_t$ rawly as an integer, since integers reflect a preset canonical order that hours influence does not necessarily follow.

One-Hot

One option is to use **One-Hot** encoding. Table 4.7 shows an example for hour one-hot codification.

$Hour_t$	Hour==1	Hour==2	Hour==3	…	Hour=24
1	1	0	0		0
3	*0*	0	1		0
5	*0*	0	0		0
…					
24	*0*	0	0		1

Table 4.7: Example of One-Hot encoding

The same goes for the day of the week. One-hot has a big problem: the dimensionality of the problem erupts, adding 33 variables when coding $Hour$ and $Weekday$. The sparsity of the dataset increases since there will be only two 1s for each timestamp (one representing the hour and one representing the weekday).

Cyclical

The alternative to one-hot is cyclical sine-cosine transformation. This transformation imposes a cycle in the structure (Fig. 4.7) of the hours of the day, making the values on the interval ends adjacent, so the 23:00 and the 00:00 of the following day take adjacent values. The same goes for the $Weekday$ (Fig. 4.8), making Sunday and Monday adjacent.

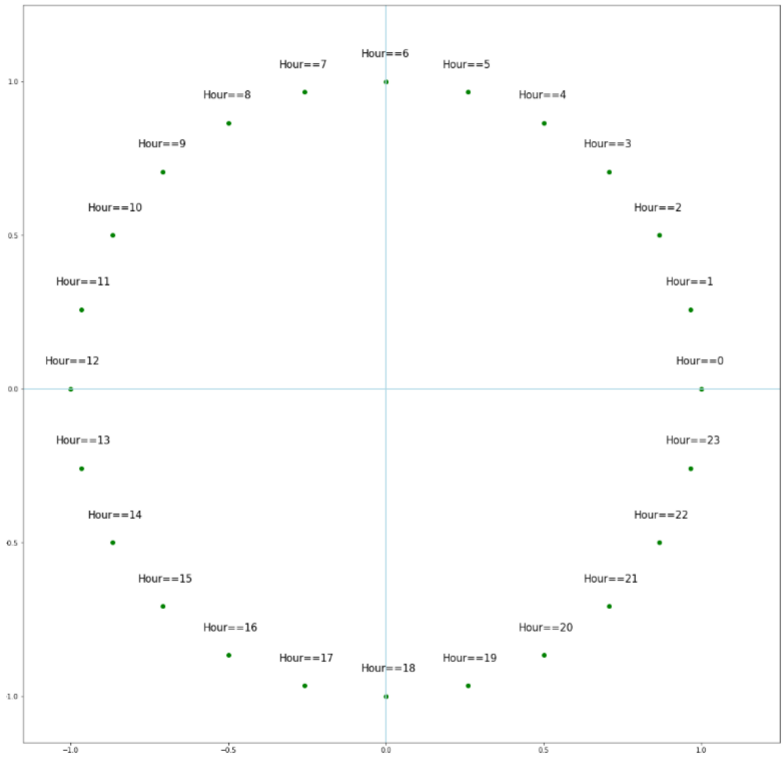


Figure 4.7: Hour Codification

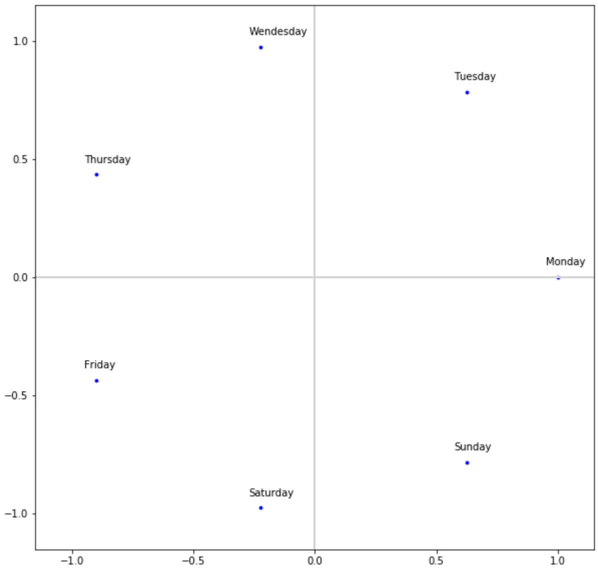


Figure 4.8: Weekday Codification

This approach has two main advantages: it only takes two variables to represent the full set of values for each possible combination of hour and weekday (4 for Hour and weekday in cyclical codification against the 33 needed for One-hot encoding) and portrays a cyclical structure in the timestamp codification, making consecutive days (like Monday and Sunday, as seen in figure 4.8 take similar values .

4.3.4 Train Test Split

The Train-Test split on figure 4.9 will be taken into consideration. As we can see, the load series has a different behaviour from November 2019 onwards, taking higher values and having more variability. Due to this, the gap before January 2020 will be used as the separator for the second train test split. This will allocate some of this *different behaviour* data into the train set. Therefore, this leads to a 87.62% — 12.48% train-test split over 9472 non missing observations

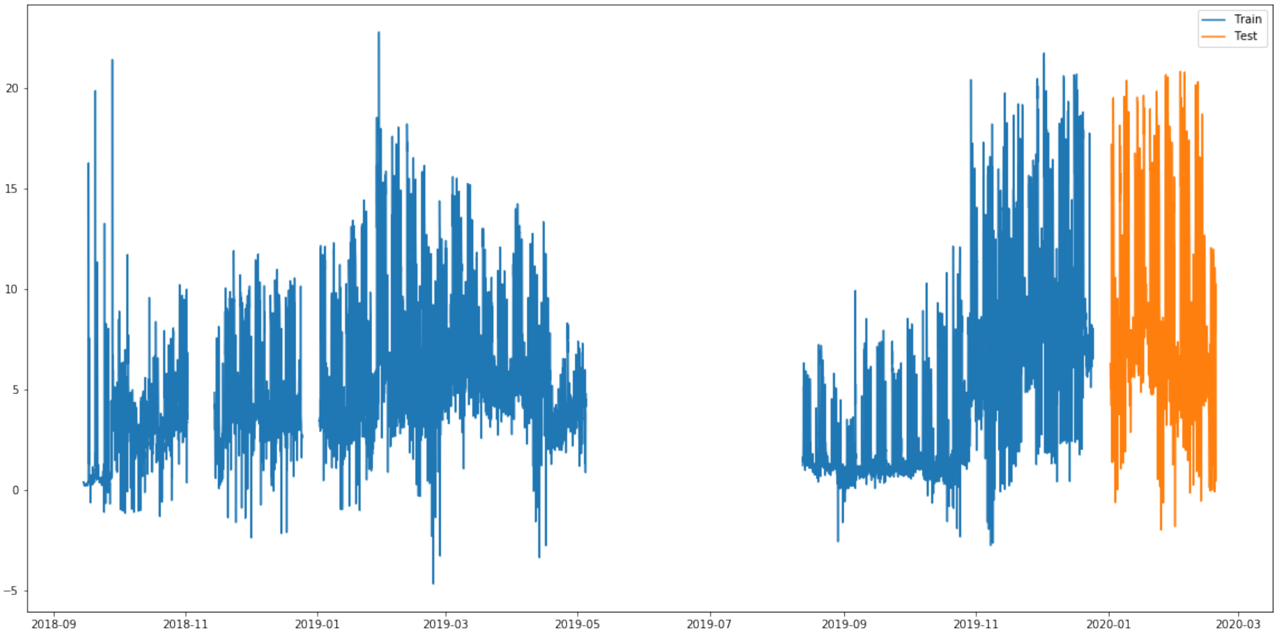


Figure 4.9: Train (blue) and Test (orange) split

4.3.5 Normalization

Normalization can boost predictive performance. In this case, the Load variable will be normalized so it has mean equal to zero and variance equal to one. (same process will be applied to weather variables when they are used). Figure 4.10 shows the series values once it's been normalized.

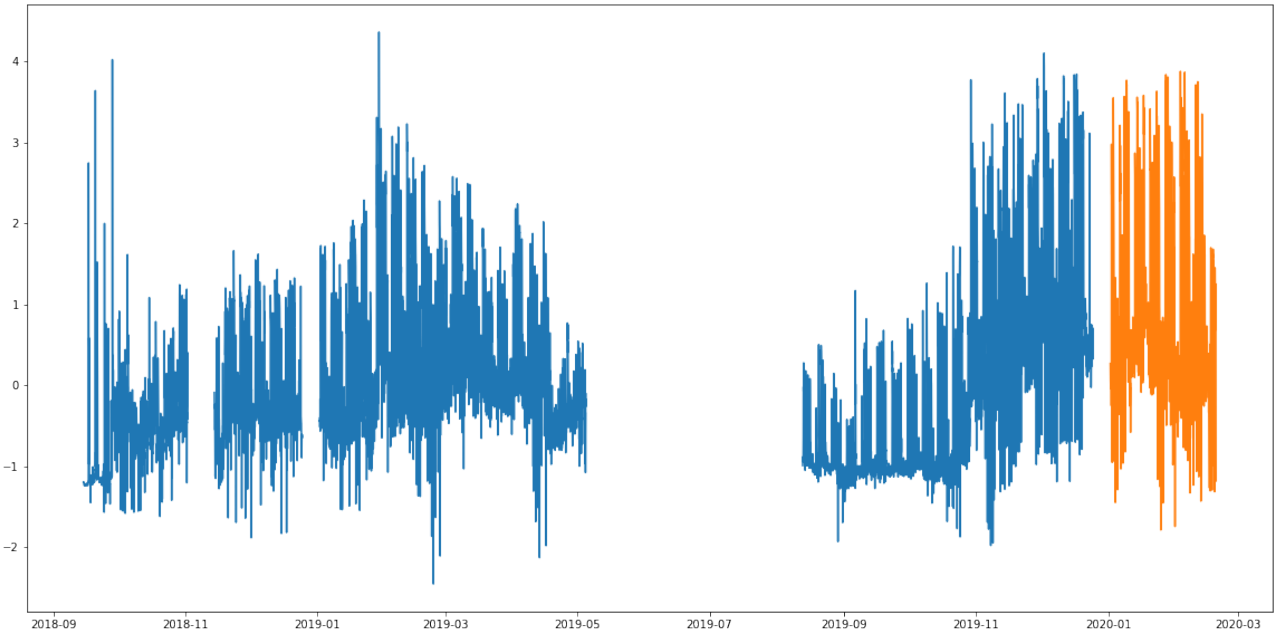


Figure 4.10: Train (blue) and Test (orange) split normalized

4.4 Testing on Benchmark: The Building Genome Project 2

In order to provide robust models, results should also be tested on a public benchmark so everyone can compare their findings. However, since the *Big Data* era started pretty recently, specially in Energy Load Management, there are not many options for this. As it was showed in table 4.1 there are not many public datasets. However, the University of Singapore alongside the University of Princeton created the Building Genome Project, and they just released the second version of this benchmark proposal [67]. As the authors state, the repository contains *3,053 energy meters from 1,636 buildings. The time range of the times-series data is the two full years (2016 and 2017) and the frequency is hourly measurements of electricity, heating and cooling water, steam, and irrigation meters.*

To see if the model adjust fine to other data, it'll be tested on the building `Wolf_education_Tammie` from that repository. Wolf buildings represent buildings from the University of Dublin, so their close location to the NUIG makes them suitable for testing and comparing. It is also an educational building with electricity reading, so this makes it our choice. The periods corresponding to winter holidays (last weeks of december) were removed since they have a different behaviour from the rest of the series. Figure 4.4 shows the train test split after preprocessing the dataset. Table 4.8 shows main statistics for the data

	μ	σ	Minimum	Maximum	Nº obs
Timestamp			2016-01-01 00:00:00	2017-12-31 23:00:00	17544
electricity_reading	82.92	21.98	0	148.485	

Table 4.8: Main statistics for the `Wolf_education_Tammie` building

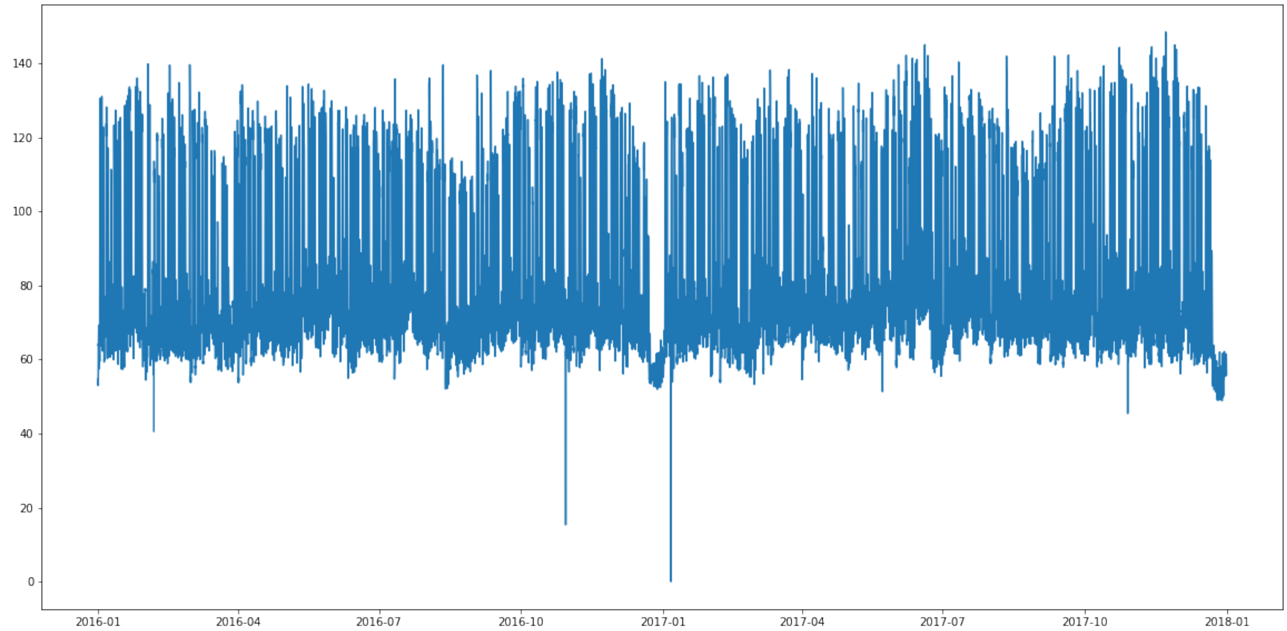


Figure 4.11: Load series for the `Wolf_education_Tammie` building

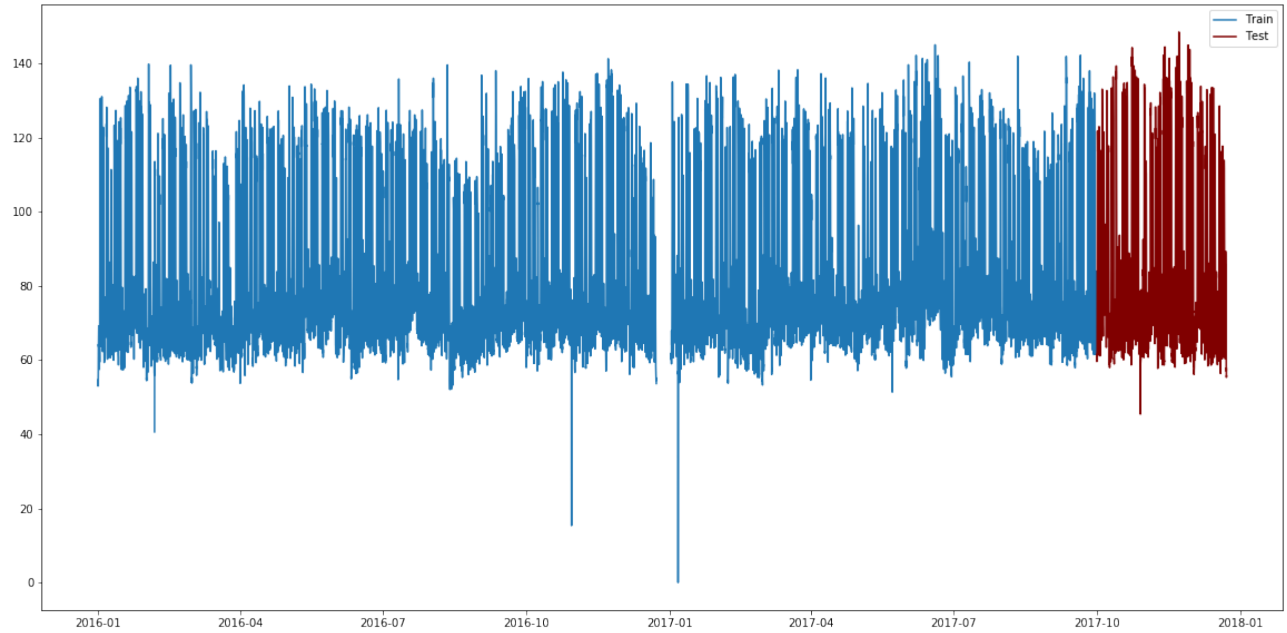


Figure 4.12: Train Test split for the `Wolf_education_Tammie` building

5. Models for One-Step ahead prediction

The main purpose of this chapter is to showcase some DNN architectures in order to see which one performs the best in the data.

5.1 Proposed models for one step ahead estimation

5.1.1 Model 1: Feed Forward Network

As Wang et al. [68] state, MLPs are often used as baseline models. However, these models incorporate modern techniques like Dropout or ReLUs as activation functions. The baseline model will be a two layered MLP, as seen in Figure 5.1.

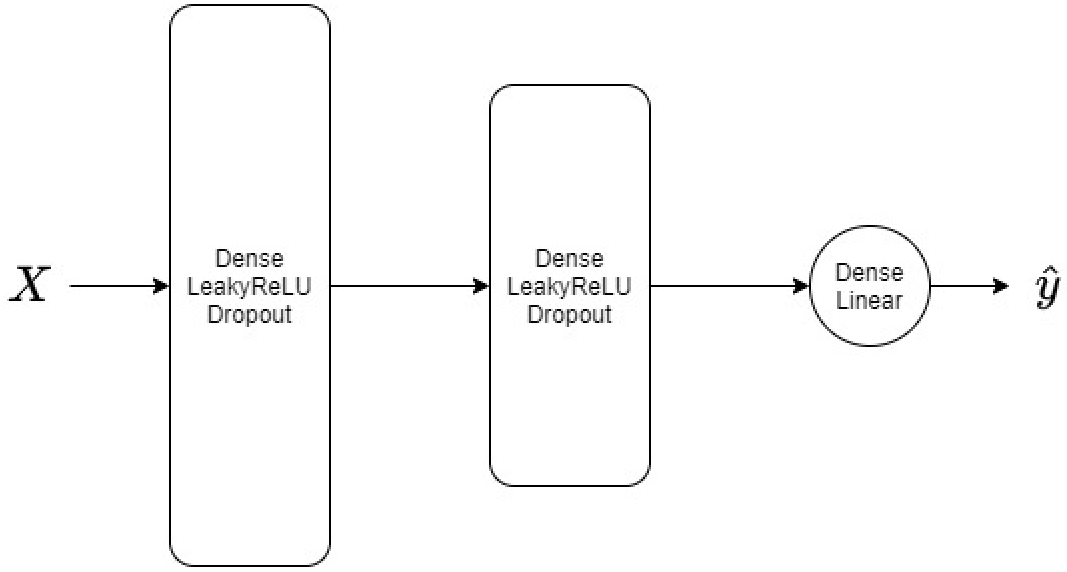


Figure 5.1: Model 1 scheme

5.1.2 Model 2: LSTM

LSTMs are the most popular architecture when it comes to time series data. Many load forecast projects use LSTMs as the core of their proposal [64] [65]. In this project a basic LSTM model will be used in order to see if it improves the FFN predictions, and check if more complex approaches (presented in the following sections) improve this model. Figure 5.2 shows the two layered LSTM network used in this project.

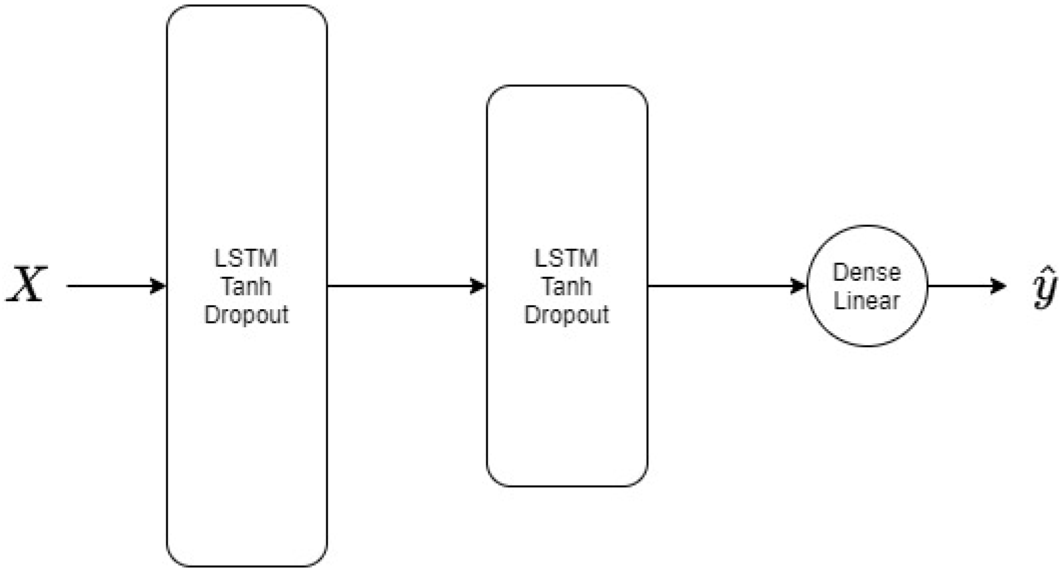


Figure 5.2: Model 2 scheme

5.1.3 Model 3: CNN

According to Ismail et al. [69], complex CNN based architectures like ResNet have been applied to Time series data. In fact, Gamboa ([70] through [69]) states that motivated by the success of these CNN architectures in these various domains, researchers have started adopting them for time series analysis. Figure 5.3 shows a two CNN-Pool block neural network, which are fed into a hidden fully conected layer before the final output.

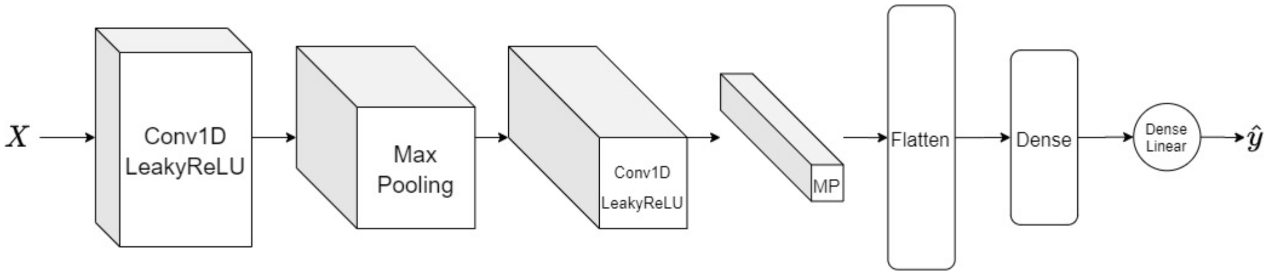


Figure 5.3: Model 3 scheme

5.1.4 Model 4: Hybrid LSTM+FFN

This architecture was proposed by Andrew Loder and Mark C. Lewis [64], which proved to outperform the FFN and the LSTM architectures in particular subsets of the data, like Winter periods. However, when working with many variables as this project's datasets have, the *Dying ReLUs* phenomenon showed up, so instead of using the ReLU activation, the Leaky ReLU is incorporated into the model. Figure 5.4 shows the architecture of this model.

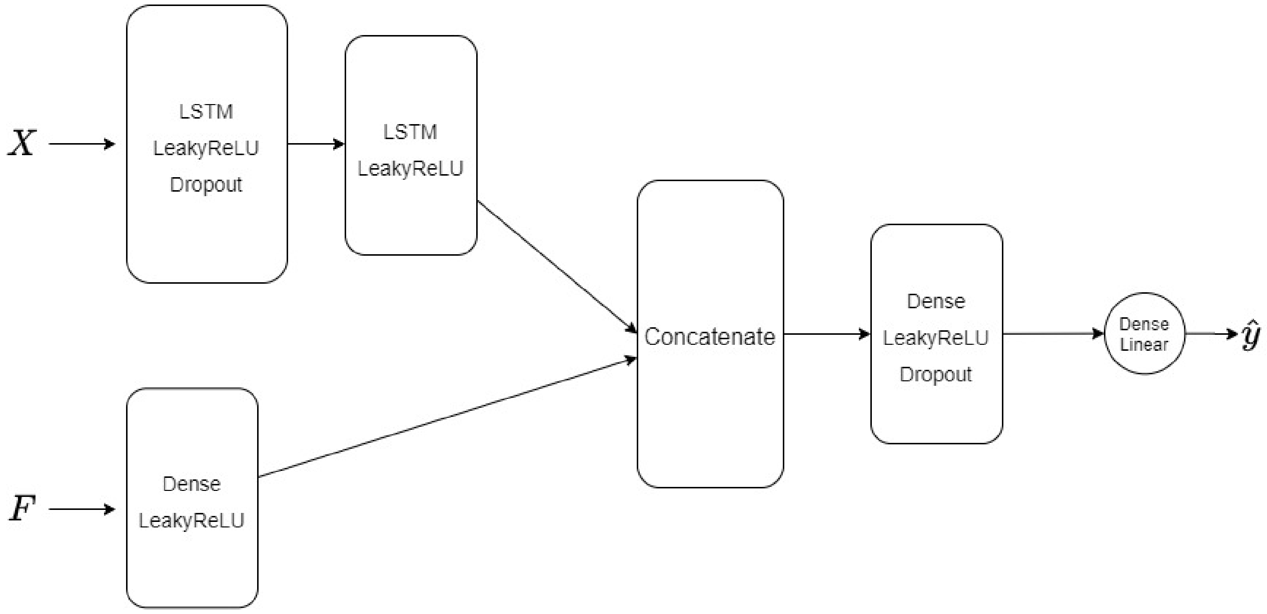


Figure 5.4: Model 4 scheme

In this architecture, the left input in Figure 5.4 takes the continuous historical values such as past Load data or weather information, which are fed into a LSTM network. On the right input, the timestamp encoding is fed into a dense layer.

Both outputs are then concatenated and fed into a dense layer, which will then be fed into the final output layer.

5.1.5 Model 5: Hybrid CNN-LSTM + FFN

In order to try to improve Model 4, a convolutional layer and a pooling layer will be added in this architecture, applied over the outputs of the LSTM layers.

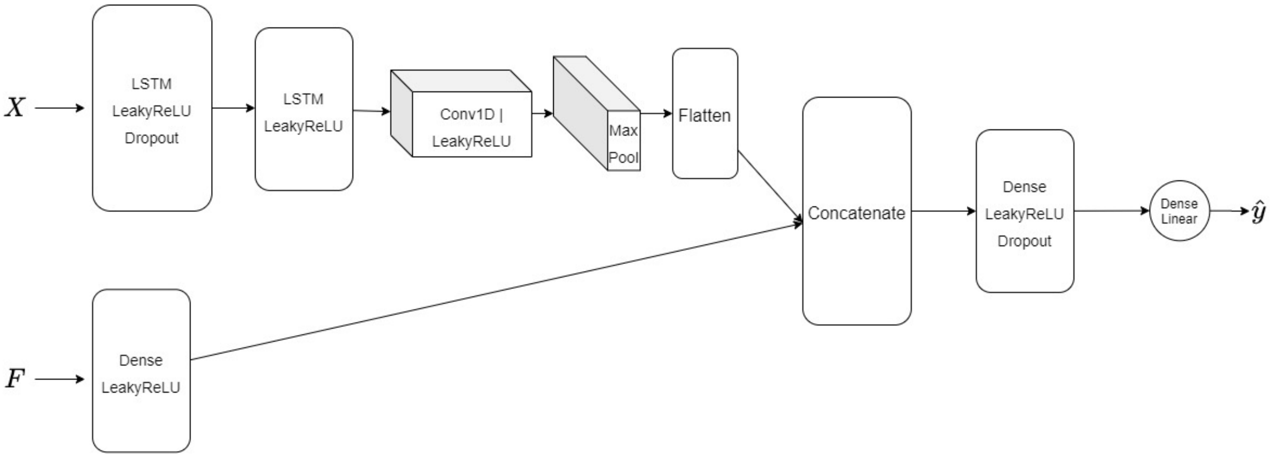


Figure 5.5: Model 5 scheme

The idea is to combine both CNNs and LSTM trying to exploit the best characteristics of each architecture.

5.1.6 Model 6: Time Distributed CNN + FFN

As an alternative to the addition of convolutions after the LSTM layers in Model 5, Model 6 is a new proposal inspired by an architecture used to process video data. A similar architecture was successfully tested by authors in [71].

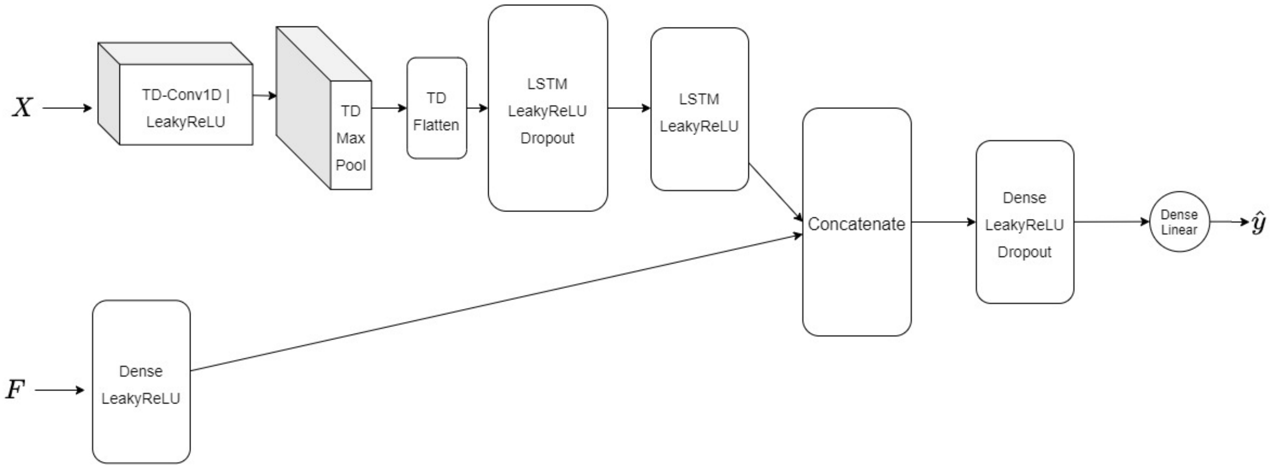


Figure 5.6: Model 6 scheme

Figure 5.7 shows how the Time Distributed CNN works. Input data is split into subsequences of 8 hours, so each day is composed of 3 subsequences. Convolutions are applied to each subsequence, and their output is then fed to a LSTM layer.

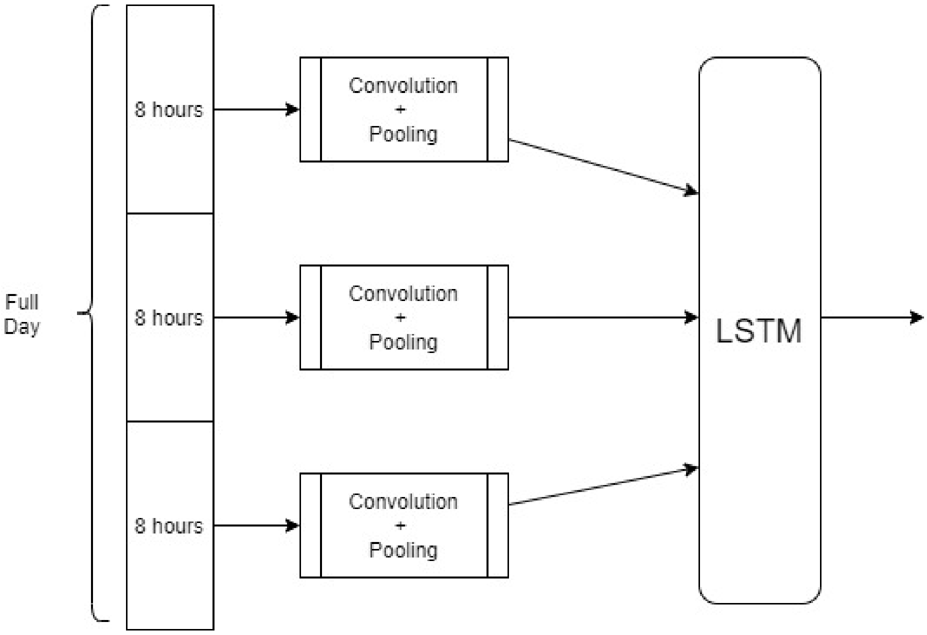


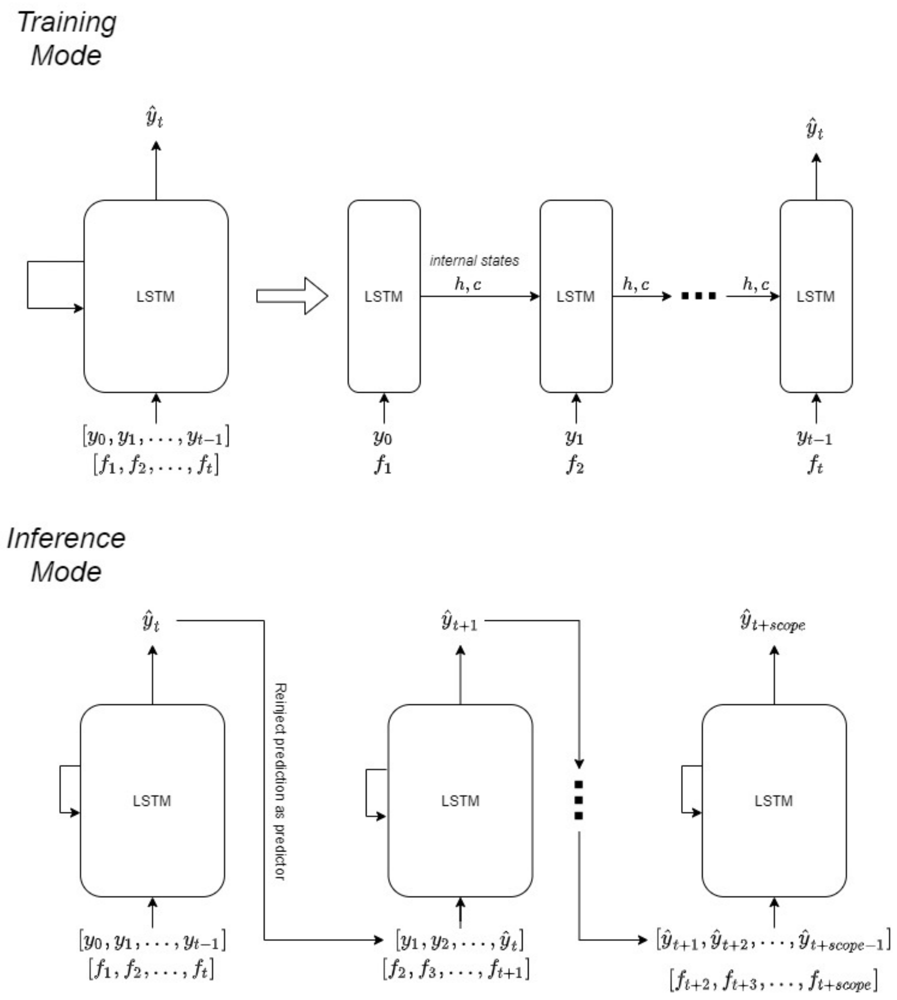
Figure 5.7: Representation of Time Distributed CNN workflow

6. Models for Sequence Prediction

In the previous chapter, models for one-step ahead prediction were showcased. However, sometimes a sequence prediction is required, in order to get an idea of how the behaviour will be for the next several hours, not just the following one. In this chapter, two main architectures are explained and tested to see what performance they give in the problem we are facing

6.1 The Many-to-One model

This architecture takes a sequence (in our case a sequence of two variables, F and Y, $[y_0, y_1, \ldots, y_{t-1}]$ as the last 60 previous load values and $[f_1, f_2, \ldots, f_t]$ as the codification for the dates) as input and outputs **one single** value as output. However, this can be adapted to sequence prediction as figure 6.1 shows. In fact, the previous chapter models were many to one, but in order to achieve a finer tuning, exclusively LSTM based architecture will be considered. The advantage of this approach against the One-to-One model is that the input is a full sequence of values, thus the network is able to capture the pattern in the input sequence.



Training Mode
$\hat{y}_t$
LSTM
LSTM
internal states
h, c
LSTM
h, c
h, c
LSTM
$\hat{y}_t$
$[y_0, y_1, \dots, y_{t-1}]$
$[f_1, f_2, \dots, f_t]$
y_0
f_1
y_1
f_2
y_{t-1}
f_t
Inference Mode
$\hat{y}_t$
$\hat{y}_{t+1}$
$\hat{y}_{t+scope}$
LSTM
Reinject prediction as predictor
LSTM
LSTM
$[y_0, y_1, \dots, y_{t-1}]$
$[f_1, f_2, \dots, f_t]$
$[y_1, y_2, \dots, \hat{y}_t]$
$[f_2, f_3, \dots, f_{t+1}]$
$[\hat{y}_{t+1}, \hat{y}_{t+2}, \dots, \hat{y}_{t+scope-1}]$
$[f_{t+2}, f_{t+3}, \dots, f_{t+scope}]$

Figure 6.1: Many to one architecture and its adaptation to sequence prediction

Nevertheless, as seen in figure 6.1, this architecture is only able to output one value. In order to predict a sequence, the prediction for the previous step must be reinjected as the latest value for the sequence. This creates a very challenging scenario, since predictions will be based on predictions. In order to solve this problem, Seq2Seq models can be applied.

6.2 The Seq2Seq model

The Secuence to Secuence Model, usually referred as seq2seq was introduced in 2014 by Sutskever et al. [72] in the Machine Translation context. The idea is to be able to generate outputs of flexible length given a set of previous values with a different length. This approach is also known as Many to Many. Previous work [65] [66] has proven the viability of this approach The baseline model will be the Encoder-Decoder model. This model consists of two submodels, the encoder and the decoder.

6.2.1 Encoder

The Encoder model takes any sequence of a fixed length and learns a hidden representation of this vector as its output. Later on, this output is fed into the decoder model, *conditioning* its output.

6.2.2 Decoder

The function of the decoder is to output the predictions based on the encoder representation of the input vector. This is known as **conditional output**; given a learned representation of the input vector, the outputs are retrieved. Figure 6.2 shows the scheme of this model.

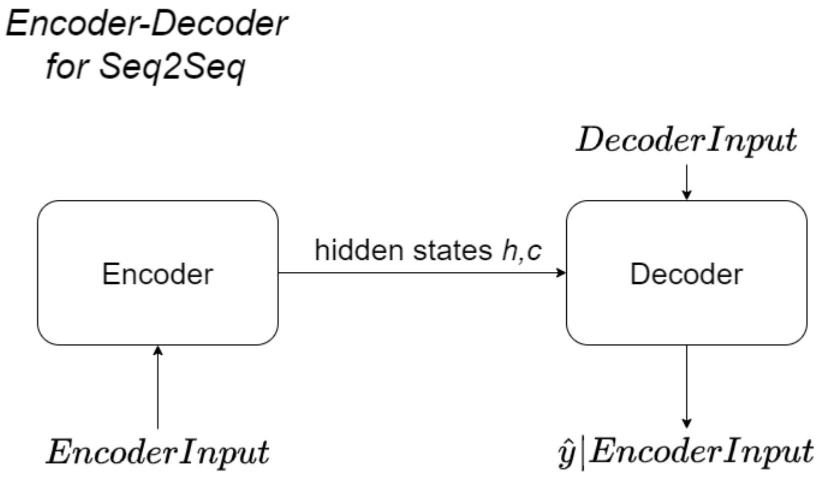


Figure 6.2: Encoder-Decoder model, showing the conditional output $\hat{y}$ given the Encoder Input

6.2.3 Training

Training the Encoder first

In [65], authors claim that pretraining the encoder first can improve model performance . The way of training it is through teacher enforcing, treating it like a One-to-one model. Figure 6.3 shows this process

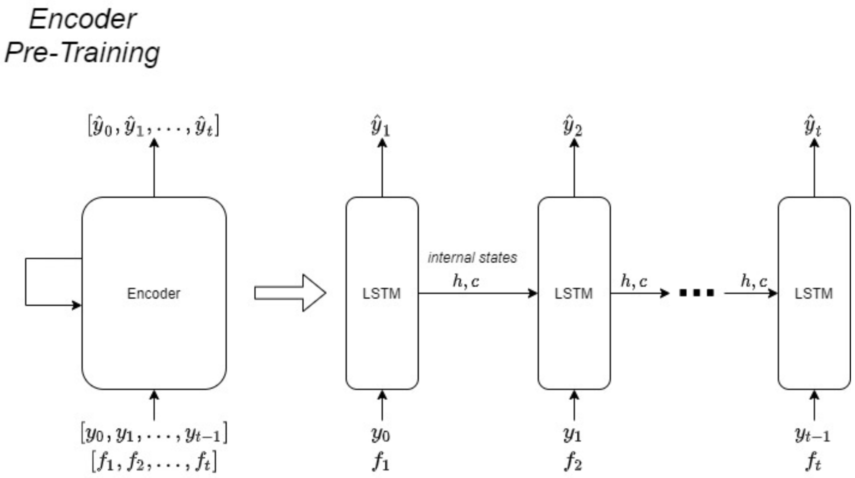


Figure 6.3: Encoder pretraining

Training the Encoder-Decoder together

Once the encoder is trained, it's plugged into the decoder for a combined training. The sequence input, consisting on both the series y and the time codification f, are fed into the encoder, which outputs its LSTM layers hidden and cell states into the decoder. The decoder takes the date codification as input and outputs the load prediction for that date given the encoder representation. Then the errors of the decoder output are backpropagated through the decoder **and** the encoder, tuning both at the same time. Figure 6.4 shows how this model is structured.

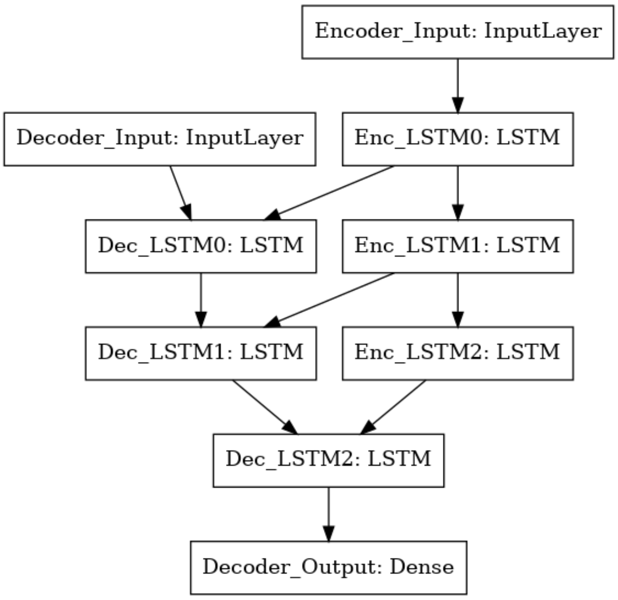
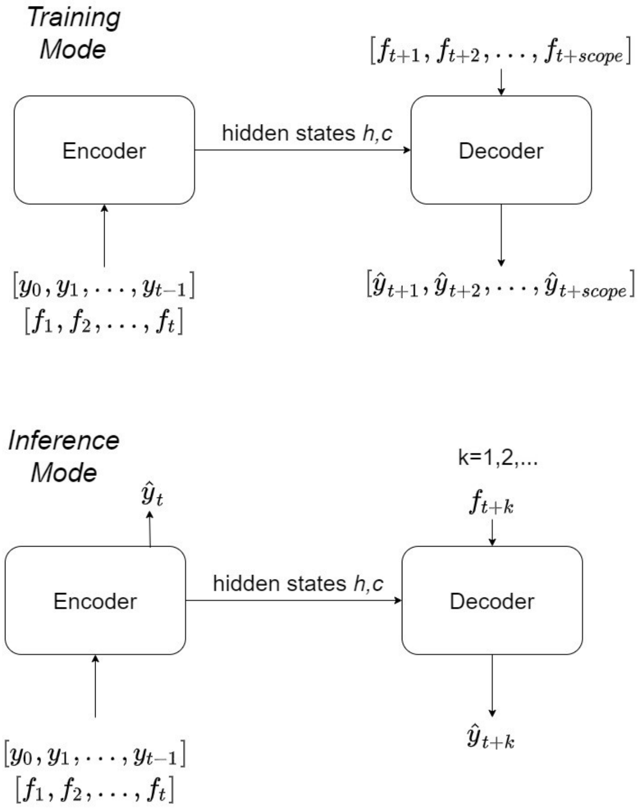


Figure 6.4: Example of a 3 layered Encoder-Decoder Keras graph

Inference

Once the Seq2Seq is trained, the inference mode works as follows:

1. The encoder takes the sequence of previous values and encodes it.
2. The encoder provides the first predicted value, $\hat{y}_t$
3. The decoder takes the encoder hidden states and sets its own hidden states to this values
4. The decoder takes the date codification for the $t+k$ date and outputs the $\hat{y}_{t+k}$ load prediction. This can be repeated as many times as it is necessary, outputing a sequence as long as it is desired (60 timesteps for our problem).



Training Mode
$[f_{t+1}, f_{t+2}, \ldots, f_{t+scope}]$
Encoder
hidden states *h,c*
Decoder
$[y_0, y_1, \ldots, y_{t-1}]$
$[f_1, f_2, \ldots, f_t]$
$[\hat{y}_{t+1}, \hat{y}_{t+2}, \ldots, \hat{y}_{t+scope}]$
Inference Mode
$\hat{y}_t$
k=1,2,...
f_{t+k}
Encoder
hidden states *h,c*
Decoder
$\hat{y}_{t+k}$
$[y_0, y_1, \ldots, y_{t-1}]$
$[f_1, f_2, \ldots, f_t]$

Figure 6.5: Encoder-Decoder combined training and inference

7. Results for One-Step prediction

7.1 Experiment Setup

In order to determine which architecture is best for this problem, hyperparameter tuning is necessary, since ANNs are strongly influenced by hyperparameters. The experiment consist in trying 6 hyperparameter combinations generated by the following code:

```
#Let p0 be the baseline hyparameters for any model

def generate_params(p0):

p1=p0.copy()
p1["nneurons"]=[nn*2 for nn in p0["nneurons"]]

p2=p0.copy()
p2["epochs"]=p0["epochs"]-10

p3=p0.copy()
p3["epochs"]=p0["epochs"]+20


p4=p0.copy()
p4["nneurons"]=[nn*2 for nn in p0["nneurons"]]
p4["epochs"]=p0["epochs"]-10

p5=p0.copy()
p5["nneurons"]=[nn*2 for nn in p0["nneurons"]]
p5["epochs"]=p0["epochs"]+20

p6=p0.copy()
p6["batch_size"]=2*p0["batch_size"]
```

```
return([p0,p1,p2,p3,p4,p5,p6])
```

The idea behind this function is to adjust the number of epochs, the number of neurons in each layer and both at the same time, where p0 are the baseline hyperparameters for any model.

The experiment setup was to try the 6 hyperparameter sets for each combination of Model and Time Codification, and repeat this 5 times, since ANNs don't provide the same set of weights in each training session due to the random weight initialization. This leads to $6models * 6hyperparameters * 2time\_codifications * 5repetitions * 2data\_scaling = 720$ models trained. This experiments will be repeated both normalizing and without normalization to check if this technique has any impact on the result.

7.2 Predictions using Load and Timestamp as predictors

Firstly, experiments for prediction using only the load and the date codification are carried. Table 7.1 shows what information goes into what part according to the model figures listed in Chapter 5. Variables under the X column are fed as a sequence, giving the last 24 values. Variables under the F column are fed as the codification for the predicted value

Model		**X**	**F**
1	*FFN*	Load, Timestamp*	
2	*LSTM*	Load, Timestamp	
3	*CNN*	Load, Timestamp	
4	*Hybrid*	Load	Timestamp*
5	*Hybrid CNN*	Load	Timestamp*
6	*TDCNN*	Load	Timestamp*

Table 7.1: Information Input for the models

*Only the previous timestamp, not the last 24
As we can see on Tables 7.2, 7.3, 7.4 and 7.5 , the best result was achieved with a LSTM-OneHot with 64 and 32 neurons in the hidden layers, training for 35 epochs and with a batch size of 64, having an RMSE of 1.595 and a CVRMSE of 0.17752. FFN, although worse, does not fall away from the LSTM.

Overall results show that normalization needs less epochs and a smaller batch. Results show also that normalization has a small (yet, positive) impact on the RMSE, specially small with LSTMs which show robustness against this data manipulation.

					RMSES	CVRMSE	Exec.Time
Model	Timecod	Nneurons	Epochs	Batch Size			
CNN	cyclical	[64, 32, 16]	35	128	1.906040	0.212018	176.558853
	one-hot	[128, 64, 32]	55	64	1.897358	0.211052	208.552271
FFN	cyclical	[128, 64]	25	64	1.863744	0.207313	83.063449
	one-hot	[128, 64]	55	64	1.619176	0.180109	85.664667
Hybrid	cyclical	[128, 64, 256, 128]	55	64	1.867077	0.207684	1267.967084
	one-hot	[128, 64, 256, 128]	55	64	1.663299	0.185017	1306.944632
HybridCNN	cyclical	[128, 64, 32, 128, 64]	25	64	1.996928	0.222128	1292.251001
	one-hot	[64, 32, 16, 64, 32]	25	64	1.677877	0.186638	1335.998616
LSTM	cyclical	[128, 64]	25	64	1.604173	0.178440	1240.101786
	one-hot	**[64, 32]**	**25**	**64**	**1.595908**	**0.177520**	1221.504934
TDCNN	cyclical	[16, 8, 8, 4, 4]	55	64	2.813699	0.312981	275.349462
	one-hot	[16, 8, 8, 4, 4]	35	64	2.654933	0.295321	279.651737

Table 7.2: Results for the best model (minimum RMSE) in 5 repetitions for each combination of Model and Time codification when Normalizing

					RMSES	CVRMSE	Exec.Time
Model	Timecod	Nneurons	Epochs	Batch Size			
CNN	cyclical	[64, 32, 16]	35	64	1.973173	0.219485	214.384258
	one-hot	[64, 32, 16]	45	128	1.961693	0.218208	254.634455
FFN	cyclical	[64, 32]	65	64	1.958445	0.217847	97.572238
	one-hot	[64, 32]	45	64	1.830539	0.203619	102.943558
Hybrid	cyclical	[64, 32, 128, 64]	45	128	1.885263	0.209707	1592.585992
	one-hot	[64, 32, 128, 64]	35	64	1.644207	0.182893	1611.872176
HybridCNN	cyclical	[64, 32, 16, 64, 32]	35	64	2.059688	0.229109	1722.782609
	one-hot	[64, 32, 16, 64, 32]	35	64	1.742375	0.193813	1654.376513
LSTM	cyclical	[128, 64]	35	128	1.629172	0.181220	1104.582622
	one-hot	**[64, 32]**	**45**	**256**	**1.607642**	**0.178826**	1074.407139
TDCNN	cyclical	[64, 64, 32, 128, 64]	45	128	1.936302	0.215384	561.597254
	one-hot	[128, 128, 64, 256, 128]	65	64	1.697899	0.188865	575.507865

Table 7.3: Results for the best model (minimum RMSE) in 5 repetitions for each combination of Model and Time codification without Normalizing

		Epochs	Batch Size	RMSES	CVRMSE	Exec.Time
Model	Timecod					
CNN	cyclical	37.0	76.8	1.934368	0.215169	177.891940
	one-hot	45.0	76.8	1.949985	0.216906	209.169296
FFN	cyclical	33.0	76.8	1.891293	0.210377	83.516319
	one-hot	39.0	64.0	1.627932	0.181083	87.378668
Hybrid	cyclical	41.0	76.8	1.876094	0.208687	1265.674493
	one-hot	35.0	76.8	1.672440	0.186033	1272.337924
HybridCNN	cyclical	33.0	64.0	2.020484	0.224748	1302.131533
	one-hot	29.0	64.0	1.707840	0.189971	1319.763845
LSTM	cyclical	33.0	64.0	1.627249	0.181007	1206.960614
	one-hot	**39.0**	**64.0**	**1.623657**	**0.180607**	1221.635948
TDCNN	cyclical	31.0	64.0	2.915580	0.324314	271.010226
	one-hot	35.0	80.0	2.705825	0.300982	276.634850

Table 7.4: Results for the average RMSE for each combination of Model and Time codification when Normalizing

		Epochs	Batch Size	RMSES	CVRMSE	Exec.Time
Model	Timecod					
CNN	cyclical	45.0	76.8	1.991736	0.221550	213.672466
	one-hot	45.0	89.6	1.980683	0.220321	254.836670
FFN	cyclical	57.0	64.0	1.969477	0.219074	97.754769
	one-hot	47.0	64.0	1.909636	0.212418	100.217364
Hybrid	cyclical	49.0	115.2	1.948947	0.216791	1590.007433
	one-hot	47.0	64.0	1.689889	0.187974	1606.031159
HybridCNN	cyclical	49.0	64.0	2.073111	0.230602	1660.540998
	one-hot	39.0	76.8	1.760850	0.195868	1660.683182
LSTM	cyclical	47.0	179.2	1.645595	0.183047	1123.512699
	one-hot	**49.0**	**179.2**	**1.617658**	**0.179940**	1127.541415
TDCNN	cyclical	51.0	76.8	1.955215	0.217488	569.087073
	one-hot	61.0	64.0	1.719208	0.191236	568.993024

Table 7.5: Results for the average RMSE for each combination of Model and Time codification without Normalizing

7.3 Predictions using Load, Weather Variables and Timestamp as predictors

The same experiments were repeated 5 times, but this time the weather variables were fed into the models aswell, in the configuration showed in Table 7.6. This time only normalized data will be considered.

Model		**X**	**F**
1	*FFN*	Load, Weather, Timestamp	
2	*LSTM*	Load, Weather, Timestamp	
3	*CNN*	Load, Weather, Timestamp	
4	*Hybrid*	Load, Weather	Timestamp
5	*Hybrid CNN*	Load, Weather	Timestamp
6	*TDCNN*	Load, Weather	Timestamp

Table 7.6: Information Input for the models with weather variables

		Nneurons	Epochs	Batch Size	RMSE	CVRMSE	Execution Time
Model	Timecod						
CNN	cyclical						
	one-hot	[64, 32, 16]	65.0	64.0	2.085355	0.231706	262.699489
FFN	cyclical	[128, 64]	35.0	64.0	2.852427	0.316936	100.249671
	one-hot	[128, 64]	35.0	64.0	3.366133	0.374015	102.776267
Hybrid	cyclical	[64, 32, 128, 64]	35.0	64.0	2.167315	0.240813	1589.026310
	one-hot	**[128, 64, 256, 128]**	**65.0**	**64.0**	**1.81020**0	0.201133	1643.041508
HybridCNN	cyclical	[64, 32, 16, 64, 32]	45.0	64.0	2.312751	0.256972	1652.556967
	one-hot	[128, 64, 32, 128, 64]	65.0	64.0	2.023087	0.224787	1666.456431
LSTM	cyclical						
	one-hot	[64, 32]	65.0	128.0	2.437552	0.270839	1082.160527
TDCNN	cyclical						
	one-hot						

Table 7.7: Minimum RMSE achieved for each combination of Model and Time codification. Blank spaces represent combinations that were not trainable due to the lack of power in the computer the experiments were carried with

As it can be seen, including weather information doesn't seem to improve prediction. In fact, the dimension of the data gets so big for some models that the machine can't handle it. Surprisingly, while the one-hot should be giving bigger dimensions, `pandas & numpy` handle it better than cyclical codification in terms of memory management. This could be due to one-hot providing highly sparse matrix (29 out of 31 values in each row are zeros), and these librarys have high efficient data structures for sparse data.

Model	Timecod	Epochs	Batch Size	RMSE	CVRMSE	Execution Time
CNN	cyclical					
	one-hot	47.0	89.6	2.253633	0.250404	258.794594
FFN	cyclical	39.0	64.0	3.714221	0.412691	100.036803
	one-hot	45.0	76.8	3.718279	0.413142	103.913929
Hybrid	cyclical	43.0	76.8	2.245731	0.249526	1594.831875
	one-hot	**45.0**	**64.0**	**1.970037**	**0.218893**	1606.658120
HybridCNN	cyclical	51.0	64.0	2.533662	0.281518	1662.550046
	one-hot	49.0	76.8	2.384882	0.264987	1608.775339
LSTM	cyclical					
	one-hot	49.0	128.0	2.537884	0.281987	1125.982756
TDCNN	cyclical					
	one-hot					

Table 7.8: Average RMSE for each combination of Model and Time codification. Blank spaces represent combinations that were not trainable due to the lack of power in the computer the experiments were carried with

7.4 Final Repetitions

As Table 7.2 indicates, the best model is the LSTM with one-hot encoding, training for 65 epochs and a batch size of 128. This set of parameters will be used in 5 final repetitions to try to get the minimum RMSE possible The minimum achieved was an RMSE of 1.593951 and a CVRMSE of 0.1772. Figure 7.1 shows the predicted time series vs the real time series for the test data.

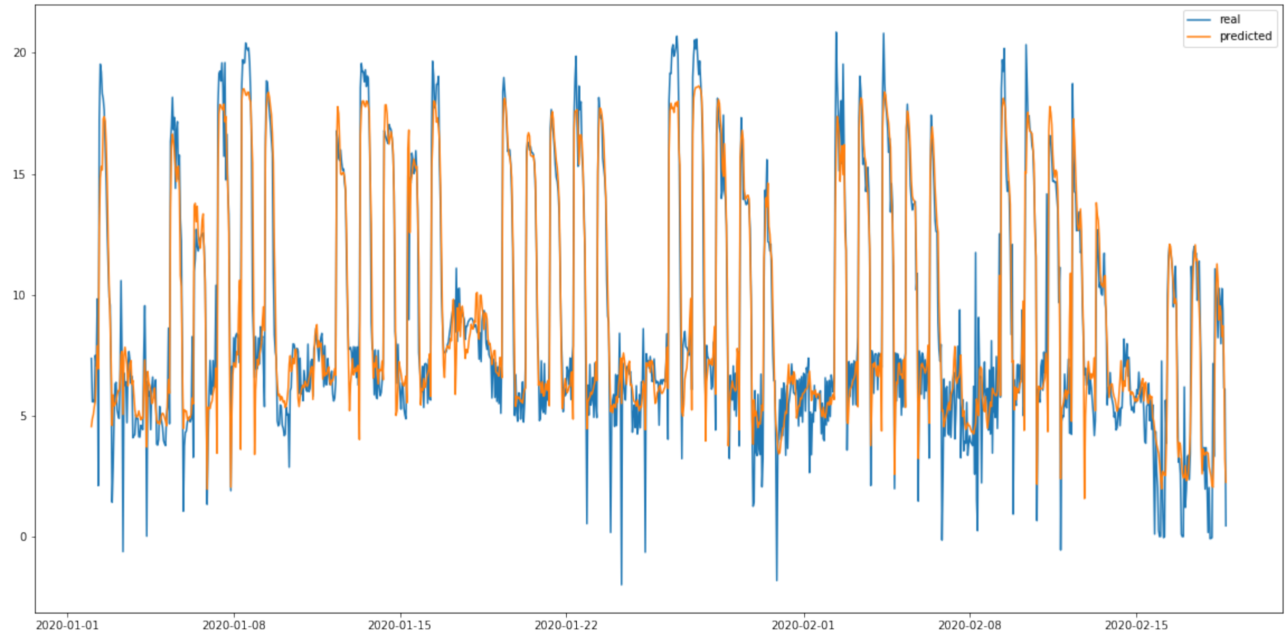


Figure 7.1: Time series prediction for the test data with the best model

7.5 Testing the best architecture on the Building Genome 2 data

The same model was trained and tested on the `Wolf_education_Tammie` building only using Load and Timestamp as predictors, achieving an RMSE of 3.46 (in the original scale) and a CVRMSE of 0.0401. Figure 7.2 presents the predicted values for this building, achieving a much better prediction than for the Alice Perry data. This low CVRMSE when using the same model reflects that the NUIG dataset is much more challenging than the available datasets, due to the big variation in the data, having a lot of outliers and many missing values.

Figure 7.2 shows how much better fitting was provided with the same model for the Building Genome 2 data than the fitting obtained in Figure 7.1.

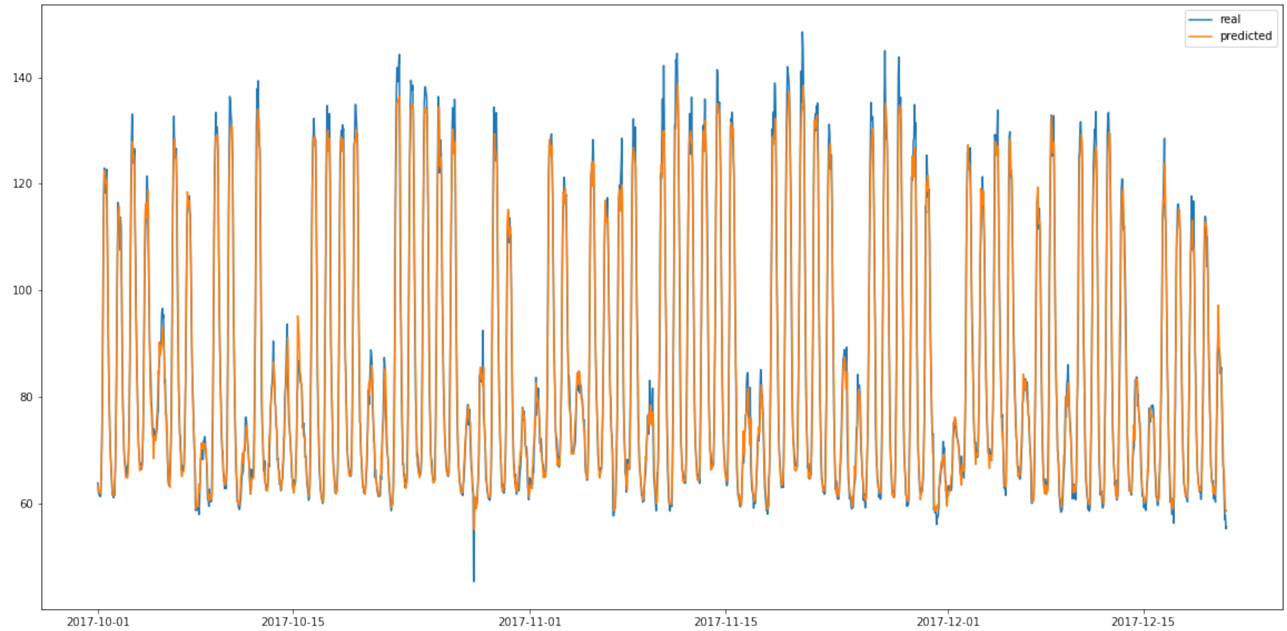


Figure 7.2: Real (blue) vs Predicted (orange) for the `Wolf_education_Tammie` building with the best model

8. Results for Sequence prediction

8.1 Experiment Setup

Multiple layered LSTM models were tested for each architecture. All models were trained using the hyperbolic tangent as an activation function. Every model was trained at 30 epochs and using a batch size of 64 for the MTO and a batch size of 128 for the Seq2Seq model. Every combination of hyperparameters was tested 5 times, due to the stochastic nature of the weight updating process, and then averaged to get an estimator of the RMSE.

The networks were tested to predict using the 60 previous observations the 60 following ones.

8.2 Results

Figure 8.1 shows the results for the five repetitions of each combination of hyperparameters. As we can see, Many-to-One models are much more stable in terms of weight calculation, having much less dispersion in the repetitions. This is probably due to the fact that the Seq2Seq model is bigger by definition, since it's composed of two submodels (although the encoder was pretrained, but by definition the seq2seq model will have more parameters).

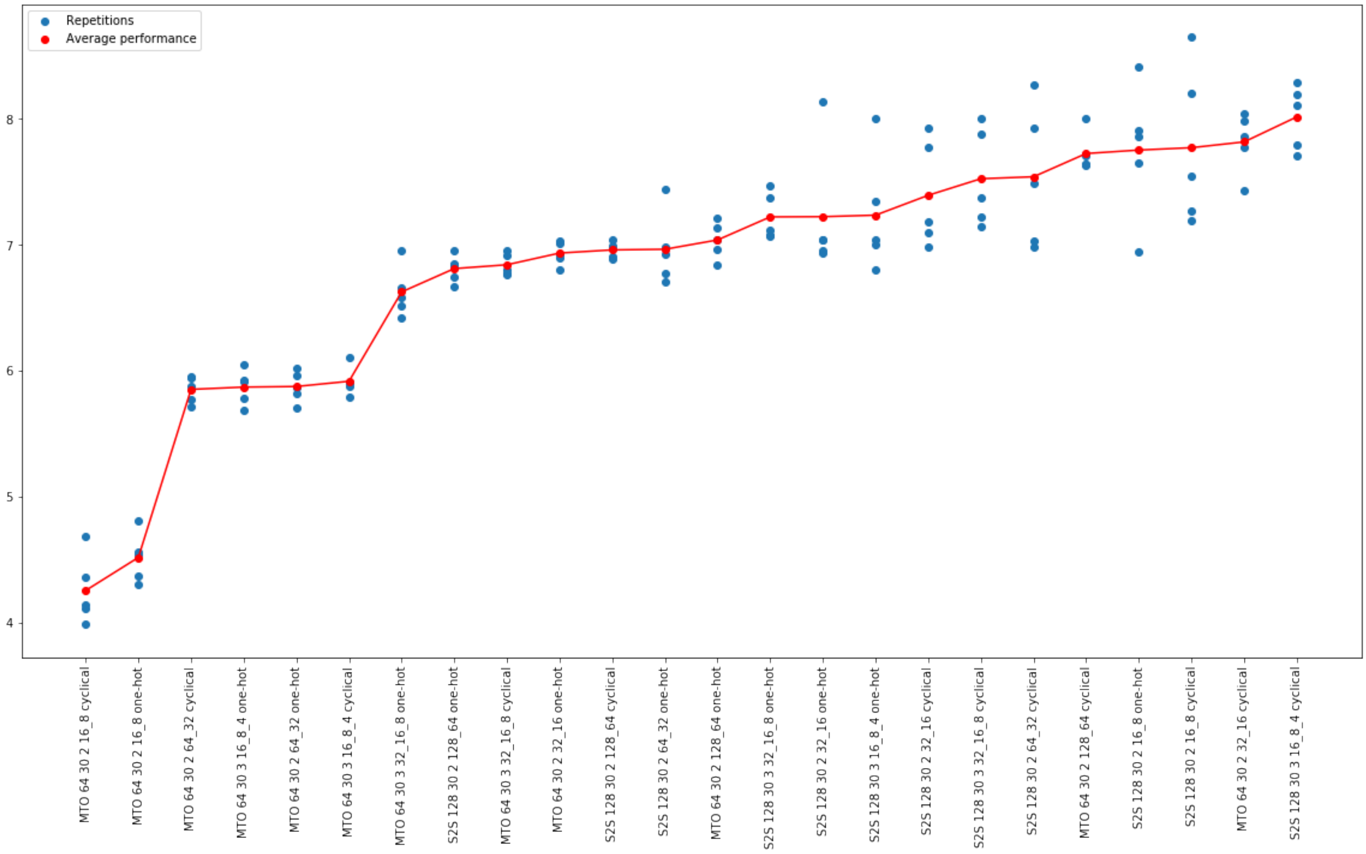


Figure 8.1: Encoder-Decoder different models Time-RMSE on Test. X-axis ticks represent the combinations for Algorithm-Batch Size-Epochs-Number of layers-Neurons per layer-Timestamp codification

						Time-RMSE	CVRMSE	Execution Time
Algo.	Batch Size	Epochs	N°Layers	Neurons	Time Cod.			
MTO	**64**	**30**	**2**	128_64	cyclical	7.725562	0.818659	1152.657166
					one-hot	7.039355	0.750309	1146.410642
				16_8	**cyclical**	**4.255968**	**0.469146**	644.930541
					one-hot	4.518206	0.488365	**634.186083**
				32_16	cyclical	7.817810	0.853784	686.002445
					one-hot	6.935232	0.760880	678.409734
				64_32	cyclical	5.852846	0.663232	822.437546
					one-hot	5.875786	0.647162	806.877860
			3	16_8_4	cyclical	5.917021	0.654249	914.205121
					one-hot	5.870928	0.644215	912.040296
				32_16_8	cyclical	6.842274	0.740326	962.612950
					one-hot	6.626617	0.714234	964.576747
S2S	**128**	**30**	**2**	**128_64**	cyclical	6.960216	0.708390	1421.530302
					one-hot	**6.811868**	**0.695569**	1410.228048
				16_8	cyclical	7.771836	0.798580	626.538593
					one-hot	7.752938	0.789764	640.614926
				32_16	cyclical	7.393806	0.753349	653.313766
					one-hot	7.224111	0.738680	660.806733
				64_32	cyclical	7.540910	0.770091	893.147289
					one-hot	6.965215	0.709630	905.810530
			3	16_8_4	cyclical	8.016221	0.820358	850.189411
					one-hot	7.235989	0.739930	847.862240
				32_16_8	cyclical	7.525386	0.770729	902.373420
					one-hot	7.222317	0.743517	936.223573

Table 8.1: Average RMSE over 5 repetitions for combinations showed in Figure 8.1, with the minimums highlighted in black

As it can be seen, the best combination of hyperparameters (on average and on minimum) is the Many-to-One two layered model with 16 and 8 neurons, with a batch size of 64, trained for 30 epochs and using cyclical encoding for the date codification. This is, in fact, the smallest network in terms of parameters, since the cyclical encoding uses only four variables to code the timestamps.

Final Repetitions

Using the best combination of hyperparameters, 5 repetitions were made in order to look for the best weight tuning. Figure 8.2 shows the sequence prediction skipping 60 observations for the best model after repetitions, which had a RMSE of 3.45 and a CVRMSE of 0.35. Figure 8.3 shows 4 different close-up day load predictions graphs.

As it can be seen, the curve fitting captures the general behaviour of the series, however, when it comes to estimate the load peaks and drops the net doesn't capture the information. The main reason for this problem could be that these peaks and drops do not correspond to a pattern and are influenced by external factors of which no record is kept or known. Another possible reason is that as seen in Figure 4.9 the test split has a different distribution from the majority of the train split, reaching higher peaks. Therefore, in order to palliate this effect more data needs to be recollected in 2020 (thus waiting until its generated) and then train with more recent data.

However, it must be taken into account that this chapter was based on predicting with predictions, which is vastly challenging compared to the previous chapter but LSTMs still managed to get a good prediction.

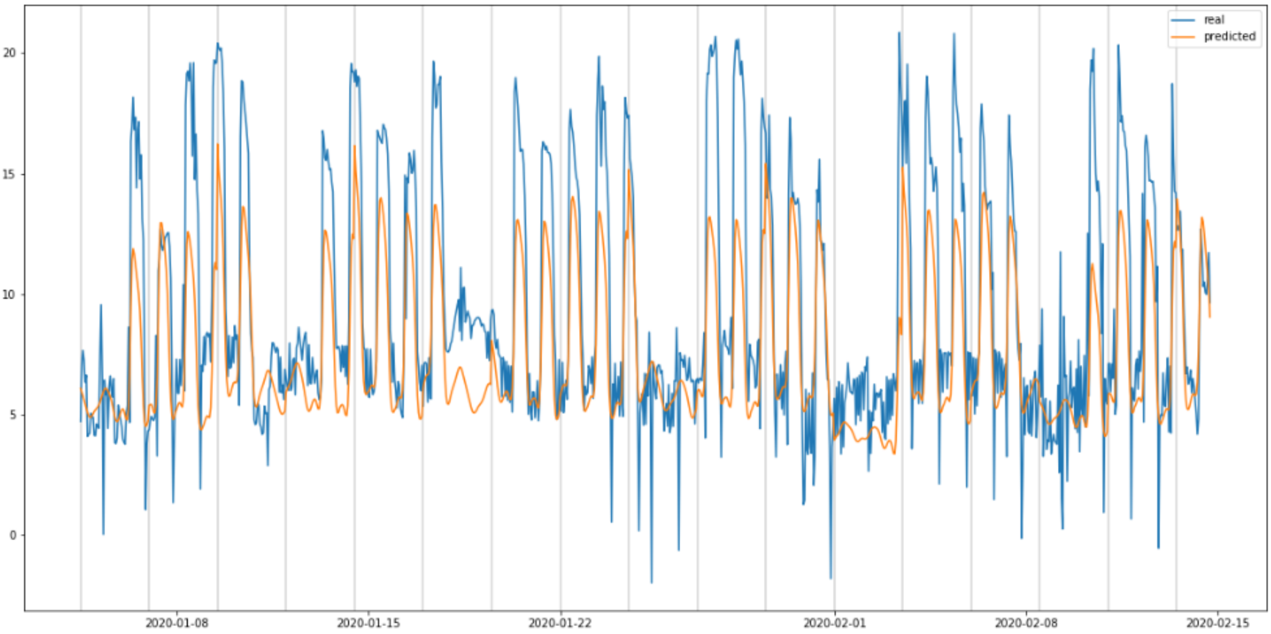


Figure 8.2: Sequence Prediction for MTO

8.3 Testing the best architecture on the Building Genome 2 data

Just as it was tested in section 4.4, the best architecture was applied to the `Wolf_education_Tammie` building data, but this time for sequence prediction. This time the model achieved a RMSE of 12.802 (in the original scale) and a CVRMSE of 0.1436. Figure 8.4 using sequences but skipping the 60 intermediates so that there is a possible comparison with the original series & 8.5 shows sequence prediction closeup for 4 different sequences. Prediction according to de CVRMSE is much better for this data too when it comes to sequence prediction.

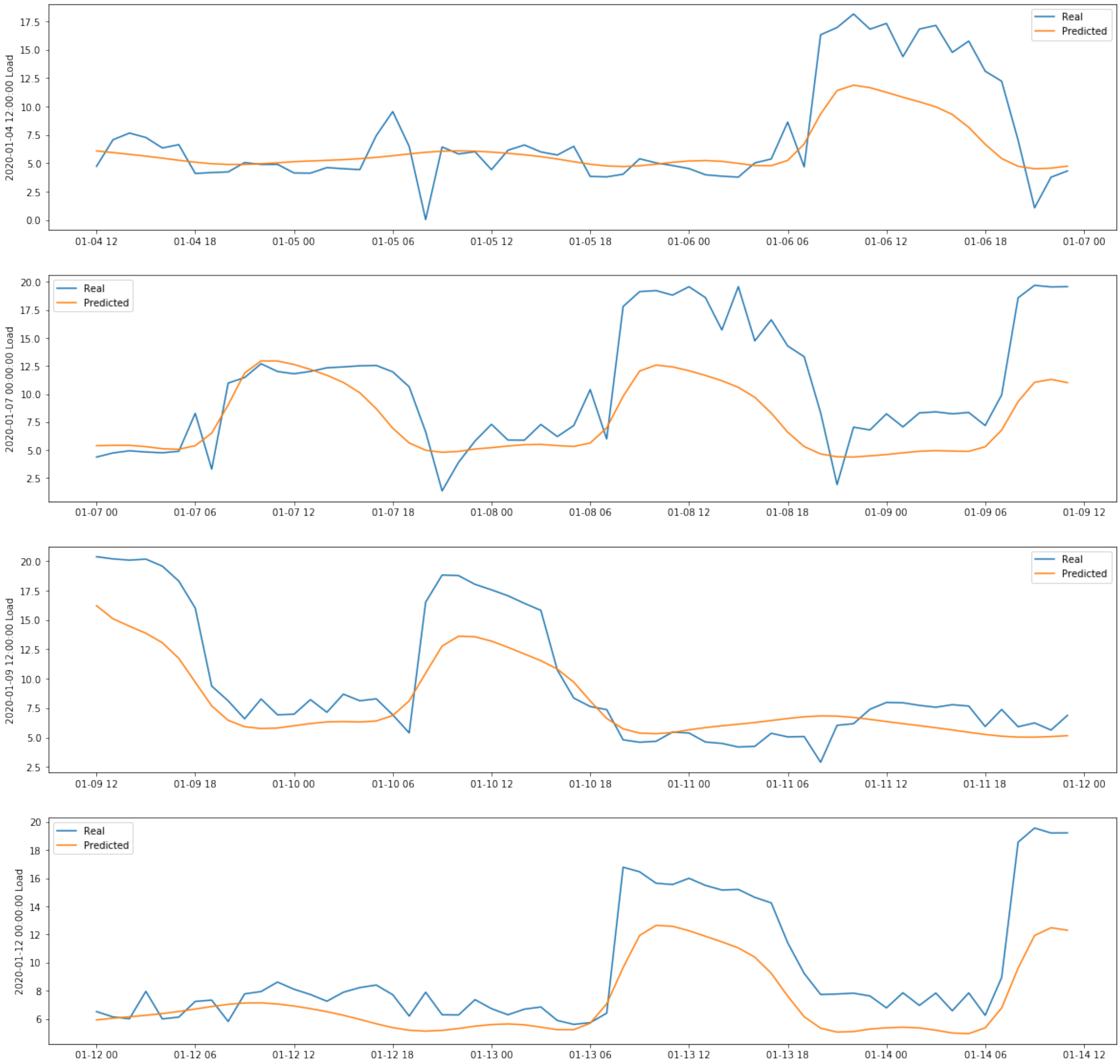


Figure 8.3: Sequence Prediction for 4 different days

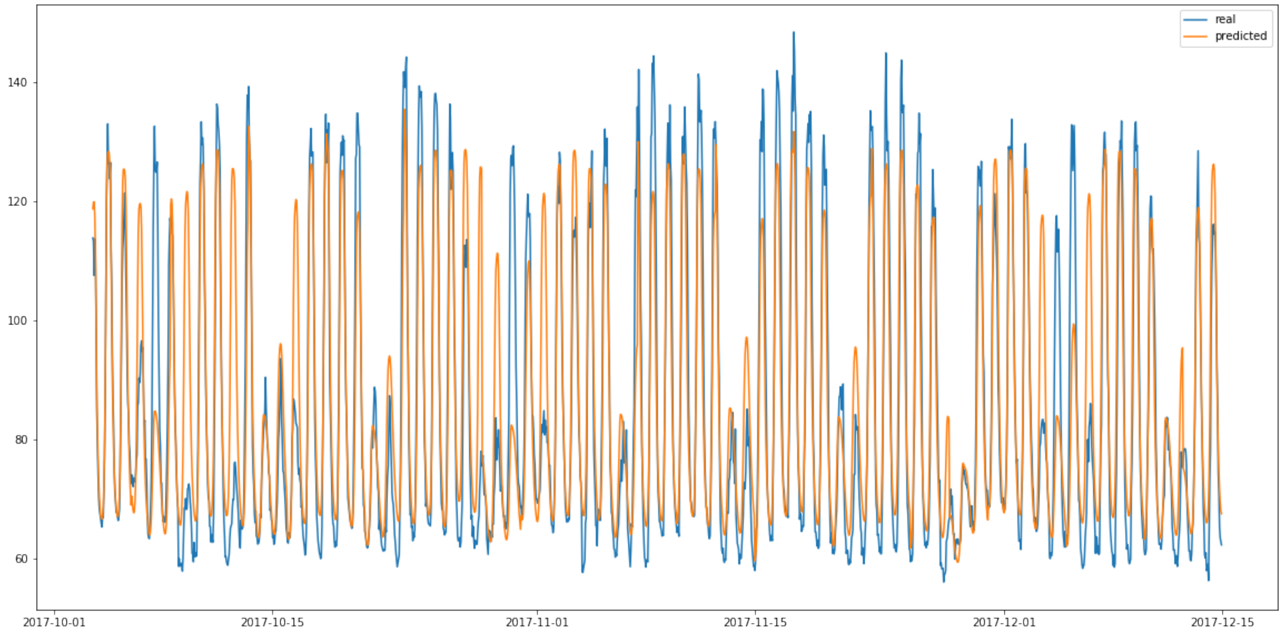


Figure 8.4: Sequence Prediction for MTO for the `Wolf_education_Tammie` building

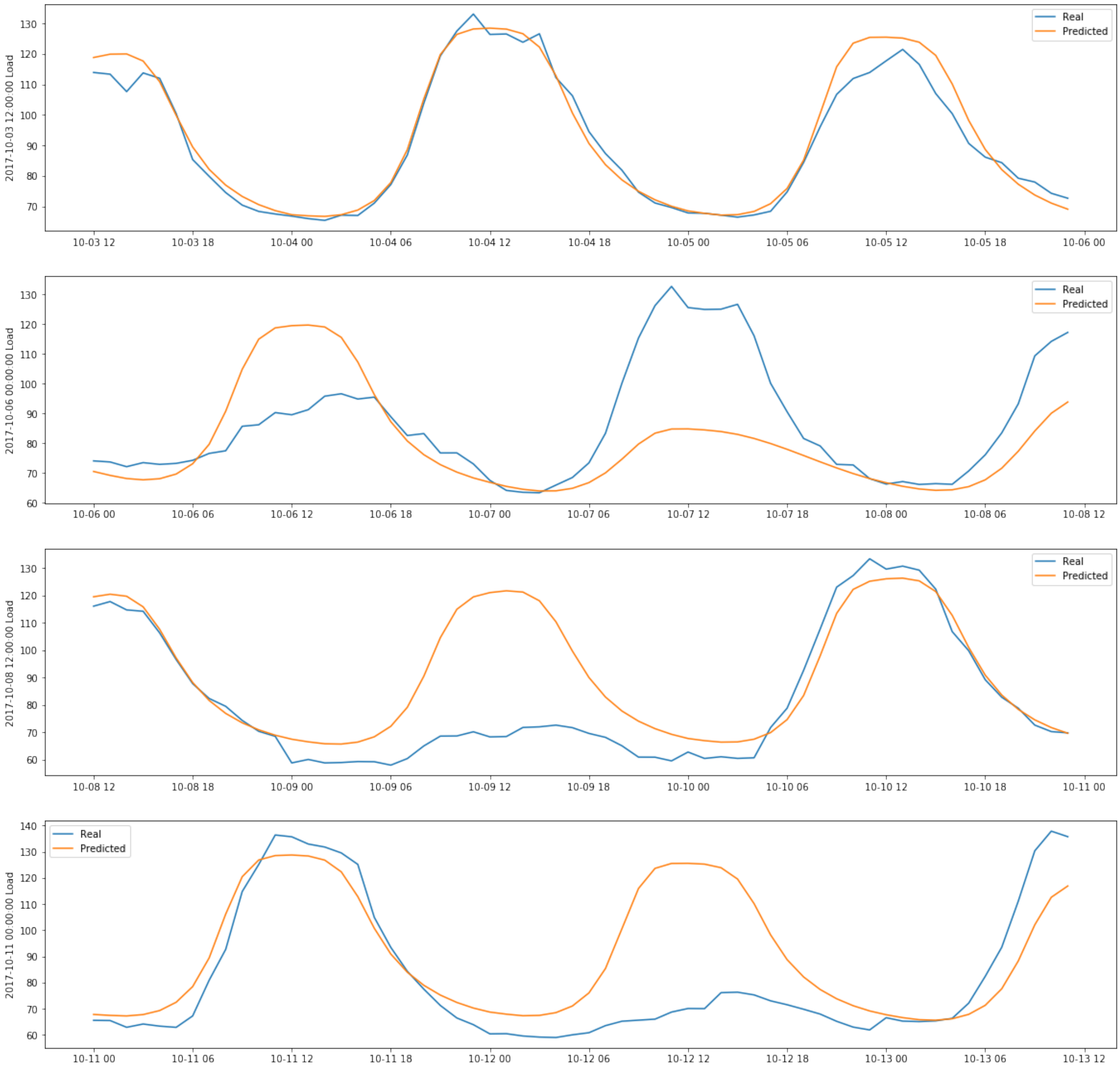


Figure 8.5: Sequence Prediction for 4 different days for `Wolf_education_Tammie` building

9. Conclusions & Further Work

This project has explored advanced deep learning techniques for time series energy load prediction, from their theoretical foundations to their performance analysis in the energy load. Work has proven the more complex architectures are not always the best option when the sample size is small. The project's main objectives have been covered succesfully, ranging from data preprocessing to time series one-step and sequence prediction for energy demand forecast in two real world examples.

9.1 Conclusions on Deep Learning

One of the main project subtasks was to gain strong knowledge and foundations in the Deep Learning Techniques used for building complex models. This goal has been achieved through a comprehensive study shown in Chapter 3. Based on that study, several deep learning architectures were selected to test their short (one-step ahead) and mid-term (i.e. sequence) forecast capabilities. The first type has been termed one-step models, while the later has been termed sequence prediction models.

9.2 Conclusions on One-step models

From advanced architectures to the baseline FFN have been studied in depth, proving the feasibility of these models in energy load prediction. The main conclusions to be drawn from these models are:

- The best model was the LSTM model, followed closely by the FFN.
- Normalization seems to slightly improve the RMSE. It also reduces the number of epochs needed but implies a smaller batch size (that is, fewer runs over the sample but smaller steps in these runs).
- Adding exogenous information (in this case, weather data) does not improve predictive performance, at least in one of the tested scenarios. However, this might be dued to the

sample size (less than 10000 observations) is small to tune models with many predictors or because the hyperparameter tuning needs to be more complex and allow to search for bigger networks.

- This applies to complex models themselves which didn't improve the LSTM model. The main reason behind this is that some specific hyperparameters like the number of layers, convolution/pooling size and stride or activation functions were not tuned, thus they might be able to achieve a better performance in the future.

9.3 Conclusions on Sequence models

S2S and MTO architectures were studied in order to see which one was more suitable for the sequence prediction problem. As it was showed, sequence prediction is not an easy task, specially when working with real world, not trivial data. The main conclusions to be drawn from these models are:

- The best architecture was the Many to One.
- Smaller models tend to work better than larger ones. No conclusions on the causes can be drawed because the main reason behind this might be either smaller models work better on sequence prediction or due to the small data sample size.
- In fact, smaller models in each familly usually work better due to the lack of data.
- Encoder-Decoder models seemed to perform poorly compared to the Many to One. This could be due to the Encoder-Decoder being, by definition, bigger than the MTO, since it has got double layers (each LSTM layer in the Encoder forces an associated layer in the Decoder).
- Even in this challenging problem, the Many to One model was able to achieve a decent performance. The main issue is that the model was not able to capture sudden peaks in the energy demand.

9.4 Conclusions on the datasets

9.4.1 Alice Perry dataset

- The dataset seems to be very challenging, having a lot of irregularities such as outliers, missing data and different behaviours. This doesn't happen in most of the datasets the literature uses (table 4.1).
- Difficult, extensive and careful pre-processing was necessary for the suitability of these data for predictive models.

- More data is required for a finer tuning. Maintenance records and other relevant information could be included into the dataset to achieve a better performance. Unfortunately, in order to collect more data, waiting until more observations are generated is necessary.

9.4.2 Building Genome 2

- The dataset is clean and accessible to everybody, which makes it extremely suitable for this field of research
- The models used on the chosen building proved that they are viable as a load prediction methodology for this kind of data.

9.5 Further Work

The advanced methodology exposed in this document opens the door for many interesting options to continue working on:

- Finer hyperparameter tuning, personalized for each architecture. An extensive literature review on hyperparameter tuning should be carried out since as this Final Degree Project showed, ANNs are extremely sensitive to them.
- To increase sample size. However as commented above, this requires waiting. A different approach could be to try **transfer learning**, but similar data must be used for this.
- To test these models against the existing results in literature. Some of the work reviewed in table 4.2 used different models for load prediction but including other techniques to improve predictive performance, such as clustering or probabilistic forecasting.
- Adapt the presented models and experiments for comparing results from the existing literature. The existing work in literature performs load prediction but adapted to hierachical prediction, load profiling using clustering, probabilistic predictions among others. In order to compare the results, the same techniques must be applied and combined with the ones presented in this Final Decree Project. There is a particular interest in comparing these techniques with the results proposed by [63] which uses the same data but including seasonal clustering and will be published shortly.

On the basis of the conclusions presented above, the project objective have been satisfactorily fulfilled and are set out in this document, completing the Final Degree Project for the Degree in Computer Engineering at the University of Valladolid.

A. Appendix

A.1 Keras Key parameters

A.1.1 Compile parameters

- `optimizer=` optimizer used for calculating weight updates based on the gradient. As mentioned, `''adam''` will be used.
- `loss=` loss function to be optimized, in this case the `''MSE''` representing the mean squared error.

A.1.2 Fit parameters

- `shuffle`: this parameter regulates shuffling the sample when keras builds the mini-batches. By default, this parameter is set to `true`. This will make consecutive samples fall in different batches, thus the gradient calculation by batch will be less precise. Hence, this parameter must be set to `false` when working with time series data.
- `epochs` & `batch size`: these two parameters regulate the number of iterations over the sample (epochs) and the mini-batch size
- `validation_split`: in order to check models before using the test partition, saving a validation sample is an option to study the evolution during the training phase. However, since hyperparameter tuning is automatized and the available sample is not a big one, this parameter will be set to zero.

A.2 Keras Layers

A.2.1 LeakyReLU

The way of incorporating this activation function layer into the model is to add a `LeakyReLU()` and setting `activation=None` in the previous layer. The key parameter is `alpha` which regulates the negative values of the activation function

A.2.2 Dense

Call with `Dense()`. Key parameters:

- `units=`number of neurons in each layer
- `activation=`activation function used in the fully conected layer

A.2.3 LSTM

Call with `LSTM`. Key parameters

- `units=`number of neurons in each layer
- `activation=`activation function used in the fully conected layer
- `input shape=` usually `Number of Samples x Sequence Length x Number of Variables`. If set to `(None,None,Number of variables)` arbitrary length for the input sequence will be accepted when predicting
- `return_sequences=` this parameter is key for the architecture nature. If set to `true`, each time one value from the input sequence is fed into the LSTM, an output is generated, setting a One-to-One situation. If set to `false`, only when the last value is processed an output will be generated, setting a Many-to-one situation

Input Shape

The LSTM input must be formatted into a 3D Tensor including *number of samples* x *timesteps* x *variables* as seen in Figure A.1 before feeding it into the LSTM layer

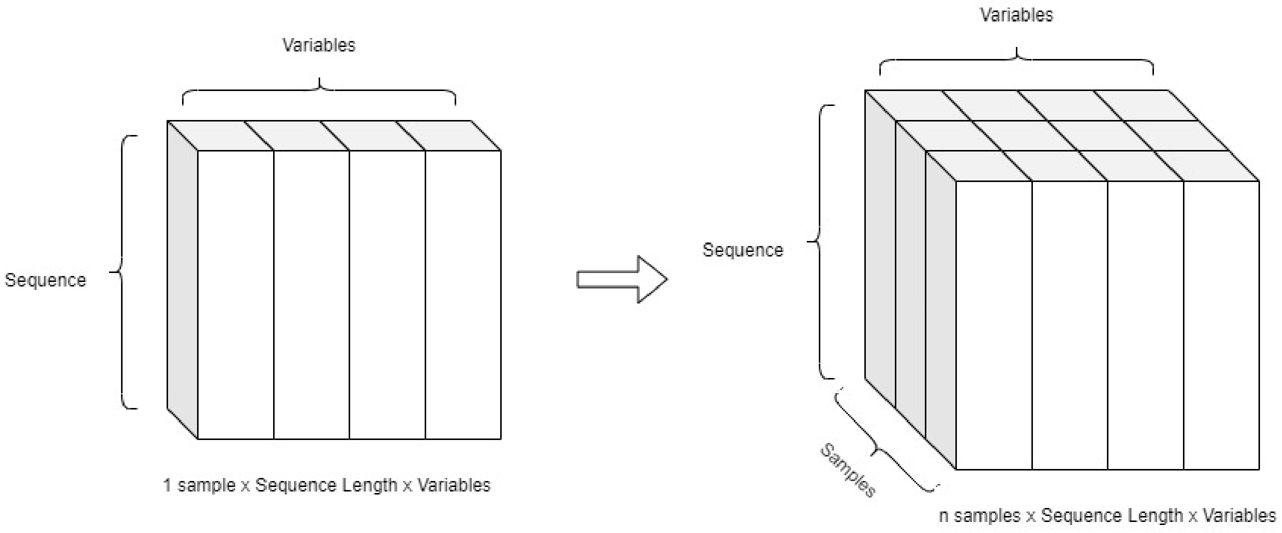

Variables
Sequence
1 sample x Sequence Length x Variables
Variables
Sequence
Samples
n samples x Sequence Length x Variables

Figure A.1: LSTM tensor shape

A.2.4 Convolutional

Call with `Conv1D()`. Key parameters:

- `filters=`number of convolutions to be applied over each input sequence
- `kernel_size=`size of each one of the convolutions
- `strides=` stride length when applying convolutions

A.2.5 Pooling

Call with `MaxPooling1D()`. Key parameters:

- `pool_size=`pooling window size
- `strides=`stride length when applying pooling
- `padding=`the input vector is padded on the ends so the output keeps the shape of the input

A.2.6 Dropout

Call with `Dropout(rate=x)`, where rate indicates the chances of dropping out a neuron.

A.2.7 Concatenate

Call with `concatenate([Output1,Output2], axis=1)`

A.3 Code

Code is available at the following repositories : Link to Repository 1 Link to Repository 2 in jupyter-notebook format. However, the Alice perry data is not of public domain, so only examples using the Building Genome data will be available there, although the code is the same except for the `df` (dataframe) assignation. In case the repository is not available please contact any of the authors.

A.4 Alice Perry Data Preprocessing

A.4.1 Auxiliar Functions

```
import pandas as pd
import numpy as np
import datetime as dt
import matplotlib.pyplot as plt
import time as clock
plt.rcParams["figure.figsize"]=(20,10)
# Aggregates data into hourly granularity.
def aggregate_by_hour(df,timecol="Timestamp"):
   df[timecol] = pd.to_datetime(df[timecol])
   time = df.Timestamp.dt
   group= df.groupby([time.year,time.month,time.day,time.hour]).mean()
   grouptime=
      pd.to_datetime([dt.datetime(year=k[0],month=k[1],day=k[2],hour=k[3]) for k
      in group.index])
   group= group.reset_index(drop="True")
   group[timecol] = grouptime
   return(group)

# Places NaN values in the negative values of the dataframe
def negative_to_na(df,feature):
   index=df["AHUs_Heating_LPHW_energy_kWh"]<0
   df[feature].loc[index]=np.nan
   return(df)


# Fills the dataset with NaN between non consecutive observations
def na_in_gaps(df,freq="H"):
   dates = pd.DataFrame()
   dates["Timestamp"] =
      pd.date_range(start=df["Timestamp"].iloc[0],end=df["Timestamp"].iloc[-1],freq=freq)

   return(pd.merge(dates,df,on="Timestamp",how="left"))


# Gets the time codification for the Timestamp
def get_time_codification(df,variables=["weekday","hour"],mode="cyclical"):

   if mode=="one-hot":
      weekday_oh = pd.get_dummies(df["Timestamp"].dt.weekday,prefix="Weekday_")
      hour_oh    = pd.get_dummies(df["Timestamp"].dt.hour,prefix="Hour_")
      month_oh   = pd.get_dummies(df["Timestamp"].dt.month,prefix="Month_")

      dict_oh    = {"weekday":weekday_oh,"hour":hour_oh,"month":month_oh}

      output     = pd.concat([dict_oh[v] for v in variables],axis=1)
      output.insert(0,"Timestamp",df["Timestamp"])

      return output

   elif mode=="cyclical":

      weekday_cos =
         pd.Series(np.cos(df["Timestamp"].dt.weekday*2*np.pi/6),name="weekday_cos")
      weekday_sin =
         pd.Series(np.sin(df["Timestamp"].dt.weekday*2*np.pi/6),name="weekday_sin")

      hour_cos   =
         pd.Series(np.cos(df["Timestamp"].dt.hour*2*np.pi/23),name="hour_cos")
      hour_sin   =
         pd.Series(np.sin(df["Timestamp"].dt.hour*2*np.pi/23),name="hour_sin")

      month_cos  =
         pd.Series(np.cos((df["Timestamp"].dt.month-1)*2*np.pi/11),name="month_cos")
      month_sin  =
         pd.Series(np.sin((df["Timestamp"].dt.month-1)*2*np.pi/11),name="month_sin")

      dict_sin   = {"weekday":weekday_sin,"hour":hour_sin,"month":month_sin}
      dict_cos   = {"weekday":weekday_cos,"hour":hour_cos,"month":month_cos}
```

```
        output     = pd.concat([dict_sin[v] for v in variables]+[dict_cos[v] for v
            in variables],axis=1)
        output.insert(0,"Timestamp",df["Timestamp"])

        return output


# Plots in two colors the chosen variable according to a boolean mask
def
    plot_masked_var(df,mask,var="Load",color="red",colorbase="#1f77b4",output=plt):
    dfcopy=df.copy()
    dfcopy[var][mask==0]=np.nan
    output.plot(df["Timestamp"],df[var],color=colorbase)
    output.plot(dfcopy["Timestamp"],dfcopy[var],color=color)
```

A.4.2 Load reading

```
#Dataset with AHUs information
df_load=pd.read_csv("../data/AlicePerry/ahusHeating.csv")

#Dataset with weather information
df_weather=pd.read_csv("../data/AlicePerry/weather/weather_station_processed2020.csv",
                sep=";").drop(["Unnamed: 0","Record"],axis=1)
df_weather["Timestamp"]=pd.to_datetime(df_weather["Timestamp"])


#Joining both datasets
df=pd.merge(df_load,df_weather,on="Timestamp",how="left")
df=df.rename(columns={df.columns[1]:"Load"})
df_with_nas=df.copy()

#Checking Nas
print(df.isna().sum())

# Plotting the load variable
plt.plot(df["Timestamp"],df["Load"])
nas=df.isna()["Batt [V]"]
plot_masked_var(df,nas)


#Plot weather variables
```

```
fig, axes = plt.subplots(10,figsize=(20,20))

for s in range(len(axes)):
   axes[s].set_ylabel(df.columns[s+2])
   axes[s].plot(df["Timestamp"],df.iloc[:,s+2],color="darkgreen")
```

A.4.3 Removing cooling and holidays periods

```
time=df["Timestamp"]
#Holiday period mask
closed=(df["Timestamp"]>dt.datetime(2018,12,25)) &
    (df["Timestamp"]<dt.datetime(2019,1,2)) |
    (df["Timestamp"]>dt.datetime(2019,12,25)) &
    (df["Timestamp"]<dt.datetime(2020,1,2))
#Cooling period (mostly summer) mask
cooling=((df["Timestamp"]<dt.datetime(2018,9,15))) |
    (df["Timestamp"]>dt.datetime(2019,5,5)) &
    (df["Timestamp"]<dt.datetime(2019,8,13))

mask=closed | cooling
plot_masked_var(df,mask,color="purple")

df["Load"].loc[mask]=np.nan
```

A.4.4 Reading Met Éireann

```
df_met_or=pd.read_csv("../data/AlicePerry/hly1875.csv").rename({"date":"Timestamp"}
                                                ,axis="columns")
df_met=df_met_or.copy() #making a copy to save original data
df_met=df_met.replace(r'^\s*$', np.nan, regex=True) # Fixing some wrong values
for col in df_met.columns[1:]:
   df_met[col] = df_met[col].astype(float) #From string to float

df_met["Timestamp"]=pd.to_datetime(df_met["Timestamp"]) # Datetime to pandas
    time class

matchmask=(df_met["Timestamp"]>=df["Timestamp"].iloc[0]) &
    (df_met["Timestamp"]<=df["Timestamp"].iloc[-1]) #Getting the same dates as
    our dataframe
```

```
df_met=df_met.loc[matchmask]
```

A.4.5 Imputing NAs with Met Éireann

Barometric Pressure

```
pressure_dif=1010.73-1013.25 #Difference of pressure between 21meters and
    0meters, since NUIG is measured at 21

df.loc[fillingindex,"BP
    [mBar]"]=df_met["msl"].loc[fillingindex].values+pressure_dif

#Sudden drop fix in one single observation
df["BP [mBar]"].iloc[5206]=df_met["msl"].iloc[5206]+pressure_dif
```

Relative Humidity

```
df.loc[fillingindex,"RH [%]"]=df_met["rhum"].loc[fillingindex].values
```

Wind Speed

```
knots_to_ms=0.514444
df.loc[fillingindex,"WindSpeed
    [m/s]"]=df_met["wdsp"].loc[fillingindex].values*knots_to_ms
```

Air Temperature

```
df.loc[fillingindex,"AirTemp [oC]"]=df_met["temp"].loc[fillingindex].values
df["AirTemp [oC]"]=df["AirTemp [oC]"]+3.5 #Error in the meter
```

Solar Radiation

```
datestofill=df["Timestamp"].loc[fillingindex]
fillingdates=datestofill+dt.timedelta(days=365) #Same period from the next year
df_weather.index=df_weather["Timestamp"]
df.loc[fillingindex,"Slr [kW/m2]"]=df_weather.loc[fillingdates,"Slr
    [kW/m2]"].values
plot_masked_var(df[4000:7000],fillingindex[4000:7000],"Slr [kW/m2]")
```

Plotting after filling

```
clean=df.isna().sum()==0
cleancols=list(df.columns[clean][1:])
fig, axes = plt.subplots(len(cleancols),figsize=(20,20))
print(cleancols)
for s in range(len(cleancols)):
   axes[s].set_ylabel(cleancols[s])
   plot_masked_var(df,fillingindex,cleancols[s],output=axes[s],color="#A70961"
              ,colorbase="darkgreen")
```

List of Figures

List of Tables

Bibliography

[1] Y. Wang et al. "Review of Smart Meter Data Analytics: Applications, Methodologies, and Challenges". In: *IEEE Transactions on Smart Grid* 10.3 (2019), pp. 3125–3148.

[2] Mahmoud A. Hammad et al. "Methods and Models for Electric Load Forecasting: A Comprehensive Review". In: *Logistics & Sustainable Transport* 11 (2020), pp. 51–76.

[3] Yanbing Lin et al. "An Ensemble Model Based on Machine Learning Methods and Data Preprocessing for Short-Term Electric Load Forecasting". In: *Energies* 10.8 (Aug. 2017), p. 1186. DOI: 10.3390/en10081186. URL: https://doi.org/10.3390/en10081186.

[4] Warren S. McCulloch and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The Bulletin of Mathematical Biophysics* 5.4 (Dec. 1943), pp. 115–133. DOI: 10.1007/bf02478259. URL: https://doi.org/10.1007/bf02478259.

[5] A G Ivakhnenko and V G Lapa. *Cybernetics and forecasting techniques.* Mod. Analytic Comput. Methods Sci. Math. Trans. from the Russian, Kiev, Naukova Dumka, 1965. New York, NY: North-Holland, 1967. URL: https://cds.cern.ch/record/209675.

[6] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1.* NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, pp. 1097–1105.

[8] Richard H. R. Hahnloser et al. "Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit". In: *Nature* 405.6789 (June 2000), pp. 947–951. DOI: 10.1038/35016072. URL: https://doi.org/10.1038/35016072.

[9] Sepp Hochreiter. "Untersuchungen zu dynamischen neuronalen Netzen". In: (Apr. 1991).

[10] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-term Memory". In: *Neural computation* 9 (Dec. 1997), pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

[11] Radford M. Neal. “Connectionist learning of belief networks”. In: *Artificial Intelligence* 56.1 (July 1992), pp. 71–113. DOI: 10.1016/0004-3702(92)90065-6. URL: https://doi.org/10.1016/0004-3702(92)90065-6.

[12] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning.* 2018. arXiv: 1811.03378 [cs.LG].

[13] George E. Dahl, Tara N. Sainath, and Geoffrey E. Hinton. “Improving deep neural networks for LVCSR using rectified linear units and dropout”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (2013), pp. 8609–8613.

[14] M. D. Zeiler et al. “On rectified linear units for speech processing”. In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing.* 2013, pp. 3517–3521.

[15] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. “Rectifier nonlinearities improve neural network acoustic models”. In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing.* 2013.

[16] Kunihiko Fukushima. “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position”. In: *Biological Cybernetics* 36.4 (Apr. 1980), pp. 193–202. DOI: 10.1007/bf00344251. URL: https://doi.org/10.1007/bf00344251.

[17] Asifullah Khan et al. “A survey of the recent architectures of deep convolutional neural networks”. In: *Artificial Intelligence Review* (Apr. 2020). ISSN: 1573-7462. DOI: 10.1007/s10462-020-09825-6. URL: http://dx.doi.org/10.1007/s10462-020-09825-6.

[18] Y. Lecun et al. “Gradient-based learning applied to document recognition”. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[19] Jeffrey L. Elman. “Finding Structure in Time”. In: *Cognitive Science* 14.2 (Mar. 1990), pp. 179–211. DOI: 10.1207/s15516709cog1402_1. URL: https://doi.org/10.1207/s15516709cog1402_1.

[20] Michael I. Jordan. “Serial Order: A Parallel Distributed Processing Approach”. In: *Neural-Network Models of Cognition - Biobehavioral Foundations.* Elsevier, 1997, pp. 471–495. DOI: 10.1016/s0166-4115(97)80111-2. URL: https://doi.org/10.1016/s0166-4115(97)80111-2.

[21] François Chollet. *Deep learning with Python.* Shelter Island, NY: Manning Publications Co, 2018. ISBN: 9781617294433.

[22] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *J. Mach. Learn. Res.* 15.1 (Jan. 2014), pp. 1929–1958. ISSN: 1532-4435.

[23] Dami Choi et al. *On Empirical Comparisons of Optimizers for Deep Learning.* 2019. arXiv: 1910.05446 [cs.LG].

[24] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization.* 2014. arXiv: 1412.6980 [cs.LG].

[25] Alberto Gasparin, Slobodan Lukovic, and Cesare Alippi. *Deep Learning for Time Series Forecasting: The Electric Load Case*. 2019. arXiv: 1907.09207 [cs.LG].

[26] Bob Hughes and Mike Cotterell. *Software project management*. London: McGraw-Hill Higher Education, 2009. ISBN: 007712279-8.

[27] Oscar Marbán, Ernestina Menasalvas, and Covadonga Fernández-Baizán. "A Cost Model to Estimate the Effort of Data Mining Projects (DMCoMo)". In: *Inf. Syst.* 33.1 (Mar. 2008), pp. 133–150. ISSN: 0306-4379. DOI: 10.1016/j.is.2007.07.004. URL: https://doi.org/10.1016/j.is.2007.07.004.

[28] P. Jokar, N. Arianpoo, and V. C. M. Leung. "Electricity Theft Detection in AMI Using Customers' Consumption Patterns". In: *IEEE Transactions on Smart Grid* 7.1 (2016), pp. 216–226.

[29] Y. Wang et al. "Sparse and Redundant Representation-Based Smart Meter Data Compression and Pattern Extraction". In: *IEEE Transactions on Power Systems* 32.3 (2017), pp. 2142–2151.

[30] H. Shi, M. Xu, and R. Li. "Deep Learning for Household Load Forecasting—A Novel Pooling Deep RNN". In: *IEEE Transactions on Smart Grid* 9.5 (2018), pp. 5271–5280.

[31] Y. Wang et al. "Clustering of Electricity Consumption Behavior Dynamics Toward Big Data Applications". In: *IEEE Transactions on Smart Grid* 7.5 (2016), pp. 2437–2447.

[32] M. Chaouch. "Clustering-Based Improvement of Nonparametric Functional Time Series Forecasting: Application to Intra-Day Household-Level Load Curves". In: *IEEE Transactions on Smart Grid* 5.1 (2014), pp. 411–419.

[33] F. L. Quilumba et al. "Using Smart Meter Data to Improve the Accuracy of Intraday Load Forecasting Considering Customer Behavior Similarities". In: *IEEE Transactions on Smart Grid* 6.2 (2015), pp. 911–918.

[34] S. Ben Taieb et al. "Forecasting Uncertainty in Electricity Smart Meter Data by Boosting Additive Quantile Regression". In: *IEEE Transactions on Smart Grid* 7.5 (2016), pp. 2448–2455.

[35] Xing Tong et al. "Cross-domain feature selection and coding for household energy behavior". In: *Energy* 107 (July 2016), pp. 9–16. DOI: 10.1016/j.energy.2016.03.135. URL: https://doi.org/10.1016/j.energy.2016.03.135.

[36] Fintan McLoughlin, Aidan Duffy, and Michael Conlon. "A clustering approach to domestic electricity load profile characterisation using smart metering data". In: *Applied Energy* 141 (Mar. 2015), pp. 190–199. DOI: 10.1016/j.apenergy.2014.12.039. URL: https://doi.org/10.1016/j.apenergy.2014.12.039.

[37] Christian Beckel et al. "Revealing household characteristics from smart meter data". In: *Energy* 78 (Dec. 2014), pp. 397–410. DOI: 10.1016/j.energy.2014.10.025. URL: https://doi.org/10.1016/j.energy.2014.10.025.

[38] X. Tong, C. Kang, and Q. Xia. “Smart Metering Load Data Compression Based on Load Feature Identification”. In: *IEEE Transactions on Smart Grid* 7.5 (2016), pp. 2414–2422.

[39] Y. Wang et al. “Deep Learning-Based Socio-Demographic Information Identification From Smart Meter Data”. In: *IEEE Transactions on Smart Grid* 10.3 (2019), pp. 2593–2602.

[40] M. Sun, I. Konstantelos, and G. Strbac. “C-Vine copula mixture model for clustering of residential electrical load pattern data”. In: *2017 IEEE Power Energy Society General Meeting*. 2017, pp. 1–1.

[41] M. Sun et al. “Probabilistic Peak Load Estimation in Smart Cities Using Smart Meter Data”. In: *IEEE Transactions on Industrial Electronics* 66.2 (2019), pp. 1608–1618.

[42] A. K. Marnerides et al. “Power Consumption Profiling Using Energy Time-Frequency Distributions in Smart Grids”. In: *IEEE Communications Letters* 19.1 (2015), pp. 46–49.

[43] O. Y. Al-Jarrah et al. “Multi-Layered Clustering for Power Consumption Profiling in Smart Grids”. In: *IEEE Access* 5 (2017), pp. 18459–18468.

[44] B. A. Høverstad et al. “Short-Term Load Forecasting With Seasonal Decomposition Using Evolution for Parameter Tuning”. In: *IEEE Transactions on Smart Grid* 6.4 (2015), pp. 1904–1913.

[45] B. Liu et al. “Probabilistic Load Forecasting via Quantile Regression Averaging on Sister Forecasts”. In: *IEEE Transactions on Smart Grid* 8.2 (2017), pp. 730–737.

[46] Nathaniel Charlton and Colin Singleton. “A refined parametric model for short term load forecasting”. In: *International Journal of Forecasting* 30.2 (Apr. 2014), pp. 364–368. DOI: 10.1016/j.ijforecast.2013.07.003. URL: https://doi.org/10.1016/j.ijforecast.2013.07.003.

[47] V. Thouvenot et al. “Electricity Forecasting Using Multi-Stage Estimators of Nonlinear Additive Models”. In: *IEEE Transactions on Power Systems* 31.5 (2016), pp. 3665–3673.

[48] James Robert Lloyd. “GEFCom2012 hierarchical load forecasting: Gradient boosting machines and Gaussian processes”. In: *International Journal of Forecasting* 30.2 (Apr. 2014), pp. 369–374. DOI: 10.1016/j.ijforecast.2013.07.002. URL: https://doi.org/10.1016/j.ijforecast.2013.07.002.

[49] Raphael Nedellec, Jairo Cugliari, and Yannig Goude. “GEFCom2012: Electric load forecasting and backcasting with semi-parametric models”. In: *International Journal of Forecasting* 30.2 (Apr. 2014), pp. 375–381. DOI: 10.1016/j.ijforecast.2013.07.004. URL: https://doi.org/10.1016/j.ijforecast.2013.07.004.

[50] Souhaib Ben Taieb and Rob J. Hyndman. “A gradient boosting approach to the Kaggle load forecasting competition”. In: *International Journal of Forecasting* 30.2 (Apr. 2014), pp. 382–394. DOI: 10.1016/j.ijforecast.2013.07.005. URL: https://doi.org/10.1016/j.ijforecast.2013.07.005.

[51] Tao Hong, Pu Wang, and Laura White. "Weather station selection for electric load forecasting". In: *International Journal of Forecasting* 31.2 (Apr. 2015), pp. 286–295. DOI: 10.1016/j.ijforecast.2014.07.001. URL: https://doi.org/10.1016/j.ijforecast.2014.07.001.

[52] Anestis Antoniadis et al. "A prediction interval for a function-valued forecast model: Application to load forecasting". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 939–947. DOI: 10.1016/j.ijforecast.2015.09.001. URL: https://doi.org/10.1016/j.ijforecast.2015.09.001.

[53] V. Dordonnat, A. Pichavant, and A. Pierrot. "GEFCom2014 probabilistic electric load forecasting using time series and semi-parametric regression models". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1005–1011. DOI: 10.1016/j.ijforecast.2015.11.010. URL: https://doi.org/10.1016/j.ijforecast.2015.11.010.

[54] Jingrui Xie and Tao Hong. "GEFCom2014 probabilistic electric load forecasting: An integrated solution with forecast combination and residual simulation". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1012–1016. DOI: 10.1016/j.ijforecast.2015.11.005. URL: https://doi.org/10.1016/j.ijforecast.2015.11.005.

[55] Stephen Haben and Georgios Giasemidis. "A hybrid model of kernel density estimation and quantile regression for GEFCom2014 probabilistic load forecasting". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1017–1022. DOI: 10.1016/j.ijforecast.2015.11.004. URL: https://doi.org/10.1016/j.ijforecast.2015.11.004.

[56] Ekaterina Mangalova and Olesya Shesterneva. "Sequence of nonparametric models for GEFCom2014 probabilistic electric load forecasting". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1023–1028. DOI: 10.1016/j.ijforecast.2015.11.001. URL: https://doi.org/10.1016/j.ijforecast.2015.11.001.

[57] Florian Ziel and Bidong Liu. "Lasso estimation for GEFCom2014 probabilistic electric load forecasting". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1029–1037. DOI: 10.1016/j.ijforecast.2016.01.001. URL: https://doi.org/10.1016/j.ijforecast.2016.01.001.

[58] Pierre Gaillard, Yannig Goude, and Raphaël Nedellec. "Additive models and robust aggregation for GEFCom2014 probabilistic electric load and electricity price forecasting". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 1038–1050. DOI: 10.1016/j.ijforecast.2015.12.001. URL: https://doi.org/10.1016/j.ijforecast.2015.12.001.

[59] Pu Wang, Bidong Liu, and Tao Hong. "Electric load forecasting with recency effect: A big data approach". In: *International Journal of Forecasting* 32.3 (July 2016), pp. 585–597. DOI: 10.1016/j.ijforecast.2015.09.006. URL: https://doi.org/10.1016/j.ijforecast.2015.09.006.

[60] J. Xie et al. "Relative Humidity for Load Forecasting Models". In: *IEEE Transactions on Smart Grid* 9.1 (2018), pp. 191–198.

[61] J. Xie and T. Hong. "Temperature Scenario Generation for Probabilistic Load Forecasting". In: *IEEE Transactions on Smart Grid* 9.3 (2018), pp. 1680–1687.

[62] Jingrui Xie and Tao Hong. "Wind Speed for Load Forecasting Models". In: *Sustainability* 9.5 (May 2017), p. 795. DOI: 10.3390/su9050795. URL: https://doi.org/10.3390/su9050795.

[63] Desirée Arias Requejo et al. "Advanced machine learning techniques for predictive maintenance of HVAC subsystems based on energy consumption prediction". In: *Building Simulation and Optimization 2020 — Accepted for publication* (2020).

[64] Mark C. Lewis Andrew Loder. "Short Term Load Forecasting Using Recurrent and Feedforward Neural Networks". In: *Proceedings of the 2019 International Conference on Artificial Intelligence* (2019), pp. 213–219.

[65] Daniel L. Marino, Kasun Amarasinghe, and Milos Manic. "Building energy load forecasting using Deep Neural Networks". In: *IECON Proceedings (Industrial Electronics Conference)*. 2016. ISBN: 9781509034741. DOI: 10.1109/IECON.2016.7793413. arXiv: 1610.09460.

[66] H. Wilms, M. Cupelli, and A. Monti. "Combining auto-regression with exogenous variables in sequence-to-sequence recurrent neural networks for short-term load forecasting". In: *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*. 2018, pp. 673–679.

[67] Clayton Miller et al. *The Building Data Genome Project 2 - Energy meter data from the ASHRAE Great Energy Predictor III competition*. 2020. arXiv: 2006.02273 [stat.AP].

[68] Zhiguang Wang, Weizhong Yan, and Tim Oates. *Time Series Classification from Scratch with Deep Neural Networks: A Strong Baseline*. 2016. arXiv: 1611.06455 [cs.LG].

[69] Hassan Ismail Fawaz et al. "Deep learning for time series classification: a review". In: *Data Mining and Knowledge Discovery* 33.4 (Mar. 2019), pp. 917–963. ISSN: 1573-756X. DOI: 10.1007/s10618-019-00619-1. URL: http://dx.doi.org/10.1007/s10618-019-00619-1.

[70] John Cristian Borges Gamboa. *Deep Learning for Time-Series Analysis*. 2017. arXiv: 1701.01887 [cs.LG].

[71] Guang Yang, HwaMin Lee, and Giyeol Lee. "A Hybrid Deep Learning Model to Forecast Particulate Matter Concentration Levels in Seoul, South Korea". In: *Atmosphere* 11.4 (Mar. 2020), p. 348. DOI: 10.3390/atmos11040348. URL: https://doi.org/10.3390/atmos11040348.

[72] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. arXiv: 1409.3215 [cs.CL].