An Overview of Deep Learning Strategies for Time Series Prediction

Rodrigo Filipe Leandro Neves

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor(s): Prof. Rui Manuel Gameiro de Castro
Dr. Silvio Miguel Fragoso Rodrigues

Examination Committee
Chairperson: Prof. Horácio Cláudio De Campos Neto
Supervisor: Prof. Rui Manuel Gameiro de Castro
Member of the Committee: Prof. Carlos Augusto Santos Silva

June 2018
For all my friends and family who have supported me during these years.
A special thank to my parents for everything.
Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.
Acknowledgments

First of all I would like to thank Prof. Rui Castro and Dr. Silvio Rodrigues for this amazing opportunity. I also want to thank for all the insights and feedback received along this work that without it none of this would have been possible.

For all the jungle people I have no words to show my gratitude for all. The help given from the first day, all the knowledge, the support and friendship from every one was amazing. Thank you.

Lastly but not least, I want to thank all of my friends who accompanied me during these years, and that made this adventure more pleasant.
Resumo

Nos últimos anos, a área de “deep learning” tem recebido muita atenção, principalmente devido aos resultados estado-de-arte obtidos em diferentes áreas como, detecção objectos, processamento de linguagem natural, processamento sequencial, entre outros. Séries temporais são um caso particular de problemas sequenciais, onde é possível aplicar modelos de “deep learning”. Dentro da área de “deep learning” é de conhecimento geral que a opção padrão a utilizar neste tipo de problemas são redes neurais recursivas. Contrariando esta tendência, resultados obtidos recentemente indicam que redes neurais convolucionais também podem ser aplicadas a problemas envolvendo séries temporais, com resultados bastante promissores. Isto levanta a seguinte questão - Quais são os melhores atributos e arquiteturas a aplicar em problemas de previsão em séries temporais? Com esta pergunta em mente, um estudo foi feito com o principal objetivo de perceber quais as arquiteturas e características mais promissoras para aplicar a problemas envolvendo séries temporais. De forma a validar este estudo, uma comparação foi feita entre as arquiteturas mais promissoras. Esta comparação foi realizada em dois problemas diferentes: previsão de energia eólica gerada e manutenção preventiva de motores turbofan. Para garantir uma comparação justa, as experiências foram conduzidas sobre as mesmas condições, onde foi possível mostrar que é possível aplicar redes neurais convolucionais, juntamente com redes neurais recursivas, a este tipo de problemas. Foi também possível mostrar que é possível aplicar vários modelos e arquiteturas a diferentes problemas, mostrando também a versatilidade presente nos diferentes modelos de “deep learning”.

Abstract

Deep learning is getting a lot of attention in the last few years, mainly due to the state-of-the-art results obtained in different areas like object detection, natural language processing, sequential modeling, among many others. Time series problems are a special case of sequential data, where deep learning models can be applied. The standard option to this type of problems are Recurrent Neural Networks (RNNs), but recent results are supporting the idea that Convolutional Neural Networks (CNNs) can also be applied to time series with good results. This raises the following question - Which are the best attributes and architectures to apply in time series prediction problems? It was assessed which is the current state on deep learning applied to time-series and studied which are the most promising topologies and characteristics on sequential tasks that are worth it to be explored. The study focused on two different time series problems, wind power forecasting and predictive maintenance. Both experiments were conducted under the same conditions across different models to guarantee a fair comparison basis. The study showed that different models and architectures can be applied on distinct time series problems with some level of success, thus showing the value and versatility of deep learning models in distinct areas. The results also showed that CNNs, together with recurrent architectures, are a viable option to apply in time series problems.

Keywords: Deep Learning, Time Series, Recurrent Neural Networks, Convolutional Neural Networks
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgments</td>
<td>vii</td>
</tr>
<tr>
<td>Resumo</td>
<td>ix</td>
</tr>
<tr>
<td>Abstract</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xv</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xvii</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>xix</td>
</tr>
<tr>
<td>Glossary</td>
<td>xxi</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Topic Overview</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Objectives</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Thesis Outline</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 Deep Learning Theory</strong></td>
<td>7</td>
</tr>
<tr>
<td>2.1 Feedforward Neural Network</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Gradient Based Learning</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Gradient Based Optimization</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Cost Function</td>
<td>10</td>
</tr>
<tr>
<td>2.2.3 Activation Functions</td>
<td>11</td>
</tr>
<tr>
<td>2.2.4 Stochastic Gradient Descent</td>
<td>13</td>
</tr>
<tr>
<td>2.2.5 Backpropagation algorithm</td>
<td>13</td>
</tr>
<tr>
<td>2.3 Convolutional Neural Network</td>
<td>14</td>
</tr>
<tr>
<td>2.3.1 Convolution Operation</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Pooling Operation</td>
<td>16</td>
</tr>
<tr>
<td>2.4 Recurrent Neural Network</td>
<td>16</td>
</tr>
<tr>
<td>2.4.1 BackPropagation Through Time</td>
<td>18</td>
</tr>
<tr>
<td>2.4.2 Long-Short Term Memory</td>
<td>21</td>
</tr>
<tr>
<td><strong>3 Deep Learning Models for Time-Series</strong></td>
<td>25</td>
</tr>
<tr>
<td>3.1 Recurrent Architectures</td>
<td>26</td>
</tr>
<tr>
<td>3.1.1 Recurrent Neural Network Architecture</td>
<td>26</td>
</tr>
</tbody>
</table>
3.1.2 Encoder-Decoder ............................................................... 26
3.1.3 Dilated Recurrent Neural Network ....................................... 28
3.2 Convolutional Architectures .................................................... 30
  3.2.1 Wavenet ................................................................. 30
  3.2.2 Temporal Convolution Block ......................................... 31
  3.2.3 Quasi Recurrent Neural Network .................................... 32

4 Methodology ................................................................. 35
  4.1 Data Normalization ........................................................... 35
  4.2 Cross-validation ............................................................. 36
  4.3 Hyperparameter Optimization .............................................. 37
  4.4 Datasets ...................................................................... 37
    4.4.1 Wind Power Generation ............................................... 37
    4.4.2 C-MAPSS Turbofan .................................................... 41

5 Results ................................................................. 45
  5.1 Wind Power Generation ....................................................... 45
    5.1.1 Wind Power Generation Hyperparameters ...................... 48
  5.2 C-MAPSS Turbofan .......................................................... 50
    5.2.1 C-MAPSS Turbofan Hyperparameters .......................... 53

6 Conclusions .............................................................. 55
  6.1 Future Work ................................................................. 56
  6.2 Contributions ............................................................... 56

Bibliography ............................................................ 57

A Pipeline ................................................................. 63
  A.1 Preprocessing Stage ........................................................ 63
  A.2 Training Stage ............................................................. 63
  A.3 Prediction Stage ........................................................... 64
  A.4 Training strategy .......................................................... 64
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Time Horizon for Wind Forecasting</td>
<td>38</td>
</tr>
<tr>
<td>4.2</td>
<td>C-MAPSS Dataset Overview</td>
<td>41</td>
</tr>
<tr>
<td>5.1</td>
<td>Wind Power Generation - Recurrent Models Hyperparameter Search Space</td>
<td>45</td>
</tr>
<tr>
<td>5.2</td>
<td>Wind Power Generation - Convolutional Models Hyperparameter Search Space</td>
<td>45</td>
</tr>
<tr>
<td>5.3</td>
<td>Wind Power Generation - Wavenet Model Hyperparameter Search Space</td>
<td>46</td>
</tr>
<tr>
<td>5.4</td>
<td>Wind Power Generation Forecasting Results</td>
<td>46</td>
</tr>
<tr>
<td>5.5</td>
<td>Hyperparameter Configuration for Winf Power Generation Forecasting Problem</td>
<td>49</td>
</tr>
<tr>
<td>5.6</td>
<td>C-MAPSS - Recurrent Models Hyperparameter Search Space</td>
<td>50</td>
</tr>
<tr>
<td>5.7</td>
<td>C-MAPSS - Convolutional Models Hyperparameter Search Space</td>
<td>50</td>
</tr>
<tr>
<td>5.8</td>
<td>C-MAPSS Problem Results</td>
<td>50</td>
</tr>
<tr>
<td>5.9</td>
<td>Results from the best model on C-MAPSS datasets</td>
<td>52</td>
</tr>
<tr>
<td>5.10</td>
<td>Hyperparameter Configuration for C-MAPSS Turbofan Problem</td>
<td>53</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Time Series Forecasting .................................................. 3

2.1 Multilayer Perceptron ..................................................... 8
2.2 Neural Network Structure ............................................... 8
2.3 Gradient Descent Algorithm ............................................ 10
2.4 Activation Functions .................................................... 13
2.5 Computational Graph Operations ...................................... 14
2.6 Comparison between ANN and CNN ................................. 15
2.7 Max Pooling Operation .................................................... 16
2.8 Recurrent Neural Network .............................................. 17
2.9 Recurrent Neural Network Unfolded Representation ........... 18
2.10 Recurrent Neural Network architectures .......................... 19
2.11 BPTT Representation in a Computational Graph ............... 20
2.12 Recurrent Neural Network Variants ................................. 22
2.13 Long-Short Term Memory Simplification ......................... 23
2.14 BPTT in Long-Short Term Memory ................................. 23

3.1 Recurrent Neural Network for Sequential Problems ............ 26
3.2 Encoder-Decoder Network ............................................... 27
3.3 Encoder-Decoder Network With Attention System ............. 27
3.4 Attention System Network ............................................... 28
3.5 Dilated Recurrent Neural Network ................................. 29
3.6 Dilated Recurrent Neural Network Simplification ............. 29
3.7 Causal Convolution ...................................................... 30
3.8 Receptive Field Size Comparison .................................... 30
3.9 Wavenet Architecture ................................................... 31
3.10 Temporal Convolution Block Architecture ....................... 32
3.11 Quasi-Recurrent Neural Network ................................... 33

4.1 Generalization Concept .................................................. 36
4.2 K-fold Cross-Validation .................................................. 36
4.3 Sliding Window Cross-Validation ..................................... 37
Nomenclature

Function Operations

\( f(x; \theta) \)  Function of \( x \) parametrized by \( \theta \).

\( \frac{df}{dx} \)  Derivative of \( f \) with respect to \( x \).

\( \frac{\partial f}{\partial x} \)  Partial derivative of \( f \) with respect to \( x \).

\( \nabla_x f \)  Gradient of \( f \) with respect to \( x \).

\( \text{arg min}_x f(x) \)  Value \( x \) that minimize function \( f \).

Operators

\( * \)  Convolution operator.

\( \otimes \)  Hadamard product - Element-wise product.

Probability

\( x \)  A random variable.

\( x \sim P \)  Random variable \( x \) has distribution \( P \).

\( p(x) \)  A probability distribution over a continuos variable \( x \).

\( \hat{p}_{\text{data}} \)  The empirical distribution defined by the training set.

\( p_{\text{data}} \)  The data generating distribution.

\( \mathbb{E}_{x \sim P}[f(x)] \)  Expectation of \( f(x) \) with respect to \( P(x) \)

\( \mathcal{N}(x; \mu, \Sigma) \)  Gaussian distribution of \( x \) with mean \( \mu \) and covariance \( \Sigma \).

Number and Arrays

\( x \)  A scalar.

\( x \)  A vector.

\( X \)  A matrix.

\( x^{(t)} \)  Value at sequence position \( t \).

\( x_i \)  \( i \)-th position of vector \( x \).
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AI</td>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td>ANN</td>
<td>Artificial Neural Network</td>
</tr>
<tr>
<td>ARIMAX</td>
<td>Auto Regressive Integrated Moving Average with eXogenous variables</td>
</tr>
<tr>
<td>ARIMA</td>
<td>Auto Regressive Integrated Moving Average</td>
</tr>
<tr>
<td>BPTT</td>
<td>BackPropagation Through Time</td>
</tr>
<tr>
<td>CNN</td>
<td>Convolution Neural Network</td>
</tr>
<tr>
<td>CV</td>
<td>Cross Validation</td>
</tr>
<tr>
<td>DRNN</td>
<td>Dilated Recurrent Neural Network</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Process Unit</td>
</tr>
<tr>
<td>GP</td>
<td>Gaussian Process</td>
</tr>
<tr>
<td>GRU</td>
<td>Gated Recurrent Unit</td>
</tr>
<tr>
<td>HMM</td>
<td>Hidden Markov Model</td>
</tr>
<tr>
<td>LSTM</td>
<td>Long-Short Term Memory</td>
</tr>
<tr>
<td>MAE</td>
<td>Mean Absolute Error</td>
</tr>
<tr>
<td>MLP</td>
<td>MultiLayer Perceptron</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>QRNN</td>
<td>Quasi Recurrent Neural Network</td>
</tr>
<tr>
<td>RNN</td>
<td>Recurrent Neural Network</td>
</tr>
<tr>
<td>RUL</td>
<td>Remaining Useful Life</td>
</tr>
<tr>
<td>SARIMA</td>
<td>Seasonal Auto Regressive Integrated Moving Average</td>
</tr>
<tr>
<td>SVM</td>
<td>Support Vector Machine</td>
</tr>
<tr>
<td>TCN</td>
<td>Temporal Convolution Network</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

Within the last years, the amount of data collected has been growing exponentially, and we should start to take advantage of that by using that data to develop new solutions that can improve our daily life [1]. The field of deep learning, that is a sub-field of machine learning, have been taking advantage of that amount of data collected in the last few years, where the recent advances in fields like vision recognition, object detection, natural language processing, generative speech systems, and others, are helping us to develop new solutions. Although, in general, people tend to talk of deep learning as a recent and new exciting technology, the truth is that the deep learning field dates back to the 1940s. Deep learning only appears to be new, because it was relatively unpopular for several years, preceding its current popularity, and because of that, has gone through many different names. It was only recently labeled as “deep learning” [2]. The field was labeled “deep learning” after the success of artificial neural networks with several layers, making them “deep”. The advances made in this area are attached with the advances in the computational resources available, and mainly with the possibility to use Graphical Processes Units (GPU) to train neural networks. GPUs are specially designed to perform matrix multiplications, thus can be used to train deep machine learning models very efficiently, speeding up algorithms up to one hundred fold [3]. This made possible to train bigger models in a fraction of the time. The field has been re-branded many times, reflecting the influence of different researchers and distinct perspectives.

The field of time series analysis has always been of fundamental importance and subject of study in statistics, and more recently, in the machine learning area. Almost every dataset with a time-stamp can be seen as a time series. For example, in the energy sector is of extreme importance to know in advance which will be the power consumed in the next days, or which will be the amount of energy generated from renewable sources. In the retail sector, every company wants to know, in advance which are going to be their sales, or how many pieces of a certain product will sell to make decisions on how they should manage their portfolio. In the heavy industry, a single machine failure can lead to enormous losses, and thereby, if that fault could be predicted in advance, could save both time and money. This is known as predictive maintenance, and can be transformed in a time series problem. There are different examples
where time series data can be found and explored to improve the outcome. The recent advances in the area of deep learning are also being applied to this field, with promising results, to different types of time series problems like time series classification, financial time series analysis or short term load forecasting [4, 5, 6].

Time series are a type of sequence modeling problem, and within the area of deep learning, the majority of studies performed about the feasibility of sequence modeling involving deep learning algorithms, are often applied to natural language processing and machine translation, where the amount of research in time series is still very scarce when compared to those problems. In order to understand which are the most promising approaches to apply in time series problems, we will conduct an empirical study about different models and methods, within the deep learning framework, that are possible to use. This study will be made on two different time series problems: forecasting and predictive maintenance, where it will be compared the most promising models found during our study. This thesis has the aim to make a fair and robust comparison across different models in the deep learning scope and understand which are the best characteristics and topologies to have in a deep learning model.

### 1.2 Topic Overview

A time series is a collection of data points that are ordered in time. A formal definition of a time series can be found in [7], where a time series is defined as a matrix $X$, where each element $x^t \in \mathbb{R}^m$, as in equation 1.1, where $x^t$ is a $m$-dimensional array, such that $x^t = [x^t_1, x^t_2, ..., x^t_m]$, that represents a collection of $m$-arbitrarily points and time $t$.

$$X = [x^1, x^2, ..., x^n]^T$$  \hspace{1cm} (1.1)

Time series analysis has the aim of extracting useful statistical properties and insights from the study of past observations, that can correctly characterize and identify the underlying structure of a series [8]. There are many advantages of this study, but one is the possibility to apply it on the time series forecasting domain. Time series forecasting is an extensively area of research and can be seen as the act of predicting unseen future values based on past values, once we know which is the time series underlying structure. A forecasting example is shown in Figure 1.1.

Time series forecasting has applications in several areas and it is one of the most studied areas of time series modeling. One can apply it to predict energy generation [9] or on the field of predictive maintenance [10]. There are other areas of applications, such as, economics, business, engineering, finance or science [11].

There is a wide variety of time series forecasting models. Two of the most known are the AutoRegressive (AR) and the Moving Average (MA) models, which are stochastic models that assume a linear combination of previous values, that follows a known statistical distribution. From the combination of AR and MA models AutoRegressive Moving Average (ARMA) and AutoRegressive Moving Integrated Average (ARIMA) appeared. They are a generalization from the previous ones. These approaches were
greatly influenced and developed by the Box-Jenkins methodology [12]. There are other generalizations that appeared from ARIMA family models, with the objective of improving some limitations. Models like, Seasonal ARIMA (SARIMA) that accounts for seasonal effects on the data or ARIMA with Exogenous explanatory variables (ARIMAX), that can introduce extra information to the forecasting problem, were introduced.

These models are widely used and studied, however this type of approaches have some problems, since they are linear models (in that the parameters are linear), which may not hold for most practical applications. To deal with these limitations, distinct methods were developed, like the vector autoregressive (VAR) [13] and autoregressive conditionally heteroscedastic (ARCH) [14] models families, which assumes a non linear combination of parameters to model the time series.

In the last few years, and following the trend of the recent advances made by machine learning models, several new approaches were developed in the area of time series. Methods like Hidden Markov Models (HMM), Gaussian Processes (GP) or Support Vector Machines (SVM), achieved good results in the forecasting and predictive maintenance domains [15, 16, 17].

Recently, Artificial Neural Networks (ANN), which are non-linear function approximator, have been receiving an increasing attention for time series forecasting. This rise of popularity is associated with the breakthroughs achieved in the area, with the successful application of Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN), to time series problems, with promising results [18]. RNNs are a type of artificial neural networks that were specially designed to deal with sequential tasks, and thus can be used on time series modeling area. They have shown promising results in the area of time series forecasting [19] and predictive maintenance [20]. CNNs that is a different type of neural network, originally designed to deal with images, are also being applied to model time series with promising results [21]. The recent advances in sequence tasks like, natural language processing [22], speech recognition [23], machine translation [24], are being adopted to time series problems [25].
1.3 Objectives

Time series problems are a big area of research across different fields which has been studied intensely in the last few decades [12, 26, 27]. This still holds true nowadays, where there is a lot of research in new approaches and methods to model time series in their different applications. Deep learning is a one of many fields where that study is being developed.

The amount of work done in the last few years regarding sequence modeling within the area of deep learning is enormous making it overwhelming, being complicated to follow all new advances and research directions. This creates a misunderstanding about which are the better approaches and directions to follow.

The aim of this thesis is to compare several deep learning sequential models that can be applied to time series problems. One of the major questions is: Which are the best attributes and architectures to apply in time series problems?. This is also dependent on the problem to solve, if it is a forecasting problem of energy consumption or if it is to predict when a machine will fail, the assumptions and target in each one are different and therefore, distinctive characteristics may lead to better results. To accomplish the main objective this comparison should be made on the most possible fair and objective way. Only through a standard comparison based on the same assumptions we can understand the advantages and disadvantages of different models.

There are different approaches and models that can be object of study and evaluate their performance across different tasks. This thesis will be focused on the ones that present an higher potential and that can be adapted to different problems. The main objective is to make a fair comparison about the performance of those models in time series problems. From this study we aim to understand which should be the default option, within the deep learning framework, to apply on time series problems.

1.4 Thesis Outline

This work will be organized in five main chapters.

- Deep Learning Theory - A theoretical approach to the deep learning framework is described in this chapter, with particular attention on time series problems.

- Deep Learning Models - This chapter will give an introduction to the models that will compared in this thesis. This introduction is also intended to explain why are the models suitable for comparison and in which problem they were applied. Other time series methods and sequence-to-sequence models will also be mentioned.

- Methodology - It will be made a clear and detailed explanation of the process taken during the experiments to get a fair comparison for the experiments. The datasets used will be introduced and it will be explained how they interconnect with the models. The performance metrics used will be introduced.
• Results - This chapter is intended to present the results for each experience made. It will be also given a comparative analysis between the different models and datasets.

• Conclusions - This final chapter has the objective to give a reflection about the results obtain and to answer to the main research questions. Future recommendations will also be given to about any future work.
Chapter 2

Deep Learning Theory

Machine learning, and especially deep learning, have been under great focus in the last few years. Machine learning is a subfield of artificial intelligence (AI), where the systems have the ability to acquire knowledge, by extracting patterns from raw data, learning from it, and then making a determination or prediction about the application that it was designed for. Deep learning, which in turn is, a subfield of machine learning, is mainly based on neural networks. Neural networks can build complex relationships, extracting the information from data, without any previous knowledge.

Feedforward neural networks, also known as MultiLayer Perceptron (MLP), are the founding principal of deep learning models. Neural networks are used as functions approximator. For example, when using a neural network to learn a classifier, \( y = f^*(x) \), that maps an input array \( x \) to a category \( y \), a MLP will approximate a function that defines the mapping \( y = f^*(x; \theta) \) between the inputs and outputs, by learning the values of the parameters \( \theta \), that gives the best function approximator. Neural networks are universal functions approximator [28] and, theoretically, they can represent any function. These models are also called feedforward neural networks, because the information flows through the network, from the input array \( x \), through the intermediate computations used to define the mapping \( f^*(x; \theta) \), until the output \( y \).

There are other types of artificial neural networks architectures like CNNs that are an artificial neural network inspired on convolution operations, or RNNs which are specially designed to deal with sequential tasks, due to the recurrent connections that they have. The building block from deep learning are going to be introduced in the next sections, as well as the others architectures.

2.1 Feedforward Neural Network

The basic construction block of all ANNs is the artificial neuron, also known as perceptron. Artificial neurons, are a mathematical function, represented by equation 2.1. It receives one or more inputs \( x_i \) and computes an output \( y \), given some weights \( w_j \). For example, let there be an input vector \( x = \{x_0, x_1, ..., x_n\} \) and its respective weight vector \( w = \{w_0, w_1, ..., w_n\} \). The output is then given by applying equation 2.1. The input \( j = 0 \) is known as the bias term, where \( x_0 = 1 \) and \( w_0 = bias \).
A feedforward neural network is constituted of artificial neurons connected, forming a network organized in layers. For example, if a network has three layers, it’s represented by equation 2.2, where the first layer, known as input layer is represented by, $f^{(1)}$. The second layer, or hidden layer, is represented by, $f^{(2)}$, and $f^{(3)}$ is the output layer. This chained structure is the most commonly one used in ANNs and it gives the depth of the model.

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$  \hspace{1cm} (2.2)

The goal of a neural network algorithm is to find the best function $f(x)$ that approximate the true function $f^\ast(x)$. To accomplish that, the network is trained with noisy approximations of $f^\ast(x)$, evaluated at different points $x$, each accompanied with the respective label $y \approx f^\ast(x)$. This process is known as supervised learning. Training examples specify what should be the output $y$ for different set of inputs $x$. The network will change the intermediate layers parameters to learn which is the mapping. This is shown in Figure 2.2.

The function $f$ introduced in 2.1, is used to introduce a non-linearity in the calculations. Without
this non-linearity it would be impossible to represent non-linear interactions between the input and the output, and the network would be limited represent linear functions. The strategy is to apply the linear mapping at some function $\Phi(x)$, instead of $x$, where $\Phi$ is a non-linear transformation. The question here is to know which mapping $\Phi(x)$ should be applied. The strategy used in neural networks is to learn $\Phi(x)$ directly from the data. The model is represented by equation 2.3. Different non-linearities can be applied, and it will be introduced in section 2.2.3.

$$y = f(x; \theta) = \Phi(x; \theta)^T W$$

(2.3)

Applying the transformation presented in equation 2.3, the parameters $\theta$ will be used to learn the best function $\Phi$ and parameters $W$, that maps the input $x$ from $\Phi(x)$, to the desired output, $y$. This task of learning and improving models by extracting features from data is extended beyond the feedforward neural networks and is applied to all kinds of models in the area of deep learning.

### 2.2 Gradient Based Learning

Designing and training a neural network is not very different from training any other machine learning model that uses gradient descent as optimization algorithm. One of the big drawbacks from ANNs is that the nonlinearity imposed by the activation function causes the loss function to become non-convex. This means that neural networks are usually trained using an iterative gradient-based optimizer, that drives the network cost function to a low value, while global convergence is not guaranteed. Gradient descent based algorithms applied to non-convex loss functions have no guarantee of convergence and are sensible to the values of the initial parameters. In neural networks the parameters that are going to be optimized are the weights and biases of the different layers.

#### 2.2.1 Gradient Based Optimization

The optimization task always involves either minimizing or maximizing a function $g(x)$ by modifying the parameters $x$. The function that is to be optimized is called the objective function, cost function, loss function or error function. The optimization problem is introduced as in equation 2.4, where we are interested in the value of $x$ that minimize function $f$.

$$x^* = \arg \min_x g(x)$$

(2.4)

The optimization problem is solved by applying the gradient descent algorithm. It is a first-order iterative optimization algorithm to find a minimum (or maximum) value of a function. The algorithm takes steps proportional to the negative of the gradient of the function to update the parameters.

For example, in a function $y = g(x)$, where both $x$ and $y$ are real numbers, the gradient of $g(x)$ gives the slope of $g$ at point $x$. Taking several steps on the negative direction of the gradient will lead the function towards a minimum, as in equation 2.5, where $\alpha$ is denoted as the learning rate and controls
the step size and has a big influence when training neural networks. An example is shown in Figure 2.3.

\[ g_{n+1}(x) \approx g(x) - \alpha g_n'(x) \] (2.5)

Some problems appears when gradient descent is used as optimization algorithm. These problems are related with the non-convex nature from the functions that are produced by neural networks. This can pose difficulties to the algorithm. Problems like a very slow rate convergence towards the minimum or to do not converge at all can appear. These types of problems can be mitigated by using techniques introduced in variations of gradient descent algorithm like Adam [29], Adadelta [30] or Adagrad [31], among others. An extensive comparison between different algorithms is given in [32].

Some problems appears when gradient descent is used as optimization algorithm. These problems are related with the non-convex nature from the functions that are produced by neural networks. This can pose difficulties to the algorithm. Problems like a very slow rate convergence towards the minimum or to do not converge at all can appear. These types of problems can be mitigated by using techniques introduced in variations of gradient descent algorithm like Adam [29], Adadelta [30] or Adagrad [31], among others. An extensive comparison between different algorithms is given in [32].

\[ H(p, q) = - \sum_i p_i \log(q_i) \] (2.6)

Several cost functions can be used when a neural network is trained and the choice of these are dependent on the problem. Most neural networks are trained using maximum likelihood principle. This means that the cost functions is simply the negative log-likelihood (or cross-entropy loss). This cost function is described as in equation 2.7, where \( E \) represents the expected value.

\[ J(\theta) = -E_{x, y \sim \hat{p}_{data}} \log p_{model}(y|x) \] (2.7)

One example of a different cost function is the mean squared error (equation 2.8a), that leads to a solution of the optimization problem, introduced by equation 2.8b. If it was possible to train on infinitely many samples from the true data generating distribution, minimizing the mean squared error cost func-
tion would give a function approximator of the mean for the probability distribution of \( y \), as in equation 2.8c.

\[
J(\theta) = \mathbb{E}_{x,y \sim p_{data}} ||y - f(x)||^2
\]  

(2.8a)

\[
f^*(x) = \arg \min_x \mathbb{E}_{x,y \sim p_{data}} ||y - f(x)||^2
\]  

(2.8b)

\[
f^*(x) = \mathbb{E}_{x \sim p_{data}} (y|x) |y|
\]  

(2.8c)

The mean absolute error it is also possible to use (equation 2.9). When the optimization problem, applied by equation 2.9b, is solved the result is a function approximator that predicts the median value of the probability distribution of \( y \). Mean squared error and maximum absolute error functions are among the most used ones when we deal with time series problems.

\[
J(\theta) = \mathbb{E}_{x,y \sim p_{data}} ||y - f(x)||
\]  

(2.9a)

\[
f^*(x) = \arg \min_x \mathbb{E}_{x,y \sim p_{data}} ||y - f(x)||
\]  

(2.9b)

One recurring theme throughout neural network design is that the gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm. Functions that saturate are an obstacle to this objective because they make the gradient become very small (or to big). In many cases, this is originated because the activation functions used to produce the output from the hidden and output units becomes saturated creating a barrier to the learning process. Therefore, activation functions must be carefully chosen.

### 2.2.3 Activation Functions

**Output units**

The choice of cost functions is tightly coupled with the output unit chosen. The output representation choice determines the cross-entropy function form. The output layer role is to provide an additional transformation to the features provided by the neural network, \( h = f(x; \theta) \). There is a wide range of functions that are possible to use, with the most known one being:

- **Linear unit** - Represented by equation 2.10, this unit is usually applied to produce the mean of a conditional Gaussian distribution \( p(y|x) = \mathcal{N}(y; \hat{y}, I) \), where maximizing the log-likelihood is equivalent to minimizing the mean squared error.
\[ f(x_i) = x_i \]  

- **Sigmoid unit** - Specially suited for classification problems with two classes where the model is represented by a Bernoulli distribution.

\[ f(x_i) = \sigma(x_i) = \frac{1}{1 + \exp(-x_i)} \]  

- **Softmax unit** - Used to represent a probability distribution over a discrete variable with \( n \) possible values.

\[ f(x_i) = \text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \]  

**Hidden units**

The design of hidden units is an active area of research within the deep learning field, and does not have many definitive guides of theoretical principles. It can be difficult to determine which one to use and it is very difficult to predict in advance which will work best. The design process consists of trial and error, intuiving that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set. Similar to the output units there is a wide range of functions that can be chosen. The most frequently use are:

- **Linear unit**

\[ f(x_i) = x_i \]  

- **Rectified linear unit**

\[ f(x_i) = \max\{0, x_i\} \]  

- **Logistic sigmoid**

\[ f(x_i) = \sigma(x_i) \]  

- **Hyperbolic tangent**

\[ f(x_i) = \tanh(x_i) \]
2.2.4 Stochastic Gradient Descent

Stochastic gradient descent is an extension of the gradient descent algorithm. A recurring problem in optimizing machine learning algorithms is that large training sets are good for generalization, but large training sets are also more computationally expensive. In the deep learning area, where it is very common to have millions of data points and the computational cost of taking the gradient descent is $O(m)$ the time to take a single step can becomes prohibitively long. The main idea behind the stochastic gradient descent is that the gradient is an expectation and thus may be approximated using small sets of samples, instead of only one. On each step of the algorithm, a mini-batch is sampled uniformly from the training data, $x = \{x_1, ..., x_m\}$. By doing this is possible to train models on training data with millions of examples, using gradient descent algorithm computed on smaller sets. The estimation of the gradient is given by equation 2.17, where $m'$ is the size of the mini-batch.

$$g = \frac{1}{m'} \sum_{i=1}^{m'} \nabla_{\theta} L(x^{(i)}, y^{(i)}, \theta)$$

(2.17)

2.2.5 Backpropagation algorithm

When training a neural network there are two steps: forward and backpropagation step. In the first one, the information will flow from the input through the output, where it is calculated the value of the cost function $J(\theta)$. The second step is where the information from that value will flow backward through the network, in order to compute the gradient. The gradient calculation is accomplished by applying the backpropagation algorithm. This is often refereed as the learning algorithm in neural networks, which
is not true because the backpropagation algorithm is only calculating the gradients that are used in the learning step, that is accomplished by applying the gradient descent optimization to update the network’s parameters.

The backpropagation algorithm is build on top of two topics: computational graphs and the chain rule of calculus. Each node of a computational graph represents a variable. This variable may be a scalar, vector, matrix, tensor, or even a variable of other type. Operations in computational graph are functions between one or more variables, where we have a set a set of allowable operations. For example, if a variable $y$ is computed by applying an operation to a variable $x$, then we draw a directed edge from $x$ to $y$. The second concept is the chain rule of calculus that enables us to compute derivatives through the graph, as is equation 2.18, where $y = g(x)$ and $z = f(g(x)) = f(y)$.

\[
\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}
\] (2.18)

This one can be generalized beyond the scalar case. Supposing that $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$ are vectors, $g$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and $f$ maps from $\mathbb{R}^n$ to $\mathbb{R}$, then the chain rule can be written as in equation 2.19, where $y = g(x)$ and $z = f(y)$. This can be written is matrix form as in equation 2.20, where $\frac{\partial y}{\partial x}$ is the Jacobian matrix of $g$.

\[
\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}
\] (2.19)

\[
\nabla_x z = (\frac{\partial y}{\partial x})^T \nabla_y z
\] (2.20)

![Diagram](image)

Figure 2.5: Computational graph using chain rule to calculate derivatives.

### 2.3 Convolutional Neural Network

CNNs are a family of neural networks that uses convolution operation in place of general matrix multiplication, in at least one of their layers. Convolutional neural networks, are a specially designed to deal with data that has a grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid with a regular sample interval, and image data, which can be seen as a 2-D grid of pixels.
2.3.1 Convolution Operation

Convolution operation is a mathematical operation between two functions that produces a third one (equation 2.21). When using deep learning terminology, the first argument, \( x \), is refereed as the input and the second argument, \( w \), as the kernel. The output is refereed as the feature map.

\[
s(t) = (x * w)(t) = \int x(a)w(t-a)\,da
\]  
(2.21)

Equation 2.22 shows a discrete convolution operation, that is used in computational operations.

\[
s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)
\]  
(2.22)

We can generalize the convolution operation to a deal with \( n \)-dimensional data. In equation 2.23 is an example of a two dimensional convolution operation, where \( I \) is the input and \( K \) represents the kernel.

\[
S(i, j) = (I * K)(i, j) = \sum_{m} \sum_{n} I(m, n)K(i-m, j-n)
\]  
(2.23)

Convolution leverages three important ideas that can help improving a machine learning system: sparse interactions, parameters sharing and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size. In shallow feedforward neural networks every input interacts with every output, but that does not happen in CNNs, as seen in Figure 2.6. For example, when processing an image, the input might have thousands or millions of pixels, but it is possible to detect small, meaningful features, such as edges, with kernels that occupy just tens or hundreds of pixels. This means that it is needed to store less parameters, which both reduces the memory requirements of the model and improves its statistical efficiency [33]. It also means that computing the output requires fewer operations.

![Figure 2.6](image)

(a) Convolution operation between two layers with a kernel size of two.
(b) Multilayer perceptron layer operation between two layers.

Figure 2.6: Comparison between a convolution network layer and a shallow network layer.
Parameter sharing (or tied weights) refers to using the same parameter for more than one function in a model. In a traditional neural network, each element of the weight matrix is used exactly once when computing the output of a layer. In convolutional networks, each member of the kernel is used at every position of the input. Taking advantage of that, convolution operations only learn one set of parameters, rather than learning a different set of parameters for every location.

In the case of CNNs, the particular form of parameter sharing causes the network to have a property called equivariance to translation. This means that, if the input changes the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function $g$ if, $f(g(x)) = g(f(x))$. In the case of convolution, if we let $g$ be any function that translates the input, meaning that, shifts it, then the convolution function is equivariant to $g$. When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation will appear in the output, with the same shift. This is also true when images are processed, where if the same pattern appears in different locations their representation will emerge in the output. Convolution operation is not equivariant to all transformations, such as changes in the scale or rotation of an input.

## 2.3.2 Pooling Operation

A typical layer of a convolutional network consists of three stages. In the first stage the layer performs several convolutions operations in parallel, where is produced distinct features maps. The second stage performs an activation function to each linear activation, introducing an non-linearity, such as a rectified linear function. In the third stage, it is used a pooling function to further modify the output of the layer. This modification introduced by the pooling function replaces the network output with a summary statistic of nearby outputs. The most used ones are the max (see Figure 2.7) and average pooling operations. There are other types of pooling operations like weighted average pooling or $L^2$ average pooling.

![Figure 2.7: Max pooling operation for a kernel size of 2x2 with a stride of one.](image)

## 2.4 Recurrent Neural Network

RNNs are a family of artificial neural networks, that have recurrent connections, as in Figure 2.8, and thereby, are specially suited for processing sequential data.
One key characteristic of RNNs is the use of shared parameters across all sequence. Instead of having a different set of parameters to process each sequence step, the same set is used across all sequence, allowing the model to generalize across sequences not seen during training. Parameter sharing is also important when a specific piece of information can occur at multiple positions within the sequence. The parameter sharing used in recurrent networks relies on the assumption that the same parameters can be used for different time steps. Equivalently, the assumption is that the conditional probability distribution over the variables at time $t + 1$ given the variables at time $t$, is stationary, meaning that, the relationship between the previous time step and the next time step does not change with time.

![Artificial neural network with recurrent connections.](image)

A good way to introduce RNNs is using computational graphs. A computational graph is a way to formalize the structure of a set of computations, such as, those involved in a RNN. It is possible to unfold a recurrent computation in a computational graph that has a repetitive structure, that typically correspond to a chain of events. Considering the classical form of a dynamical system, represented by equation 2.24, where $s(t)$ is the system as time $t$. This is a recurrent equation, since the state at time $t$ depends on the previous time step $t - 1$. It is possible to unfold this system in a directed acyclic computational graph by recursively applying equation 2.24.

$$ s(t) = f(s(t-1); \theta) \quad (2.24) $$

For a RNN the principle is exactly the same, but instead of having a state $s(t)$, RNNs have an hidden state $h(t)$ and, usually, an external signal $x(t)$ (input), as shown in equation 2.25. RNNs usually have addition layers in the input and output to add additional transformations to the data. An example of a unfolded RNN is given in Figure 2.9.

$$ h(t) = f(h(t-1), x(t); \theta) \quad (2.25) $$

When the recurrent network is trained to perform a task that requires predicting the next step based on previous values, the network typically learns to use the hidden state, $h(t)$, as a summary of task-relevant aspects of the past sequence inputs, up to the current sequence $t$. This summary is usually lossy, since it maps an variable sequence length, to a single output vector (equation 2.26), where $g(t)$ is the representation of the unfolded recurrence.
This process has two advantages. The first one is that independently of the sequence length, the hidden state will always have the same size because it is expressed as transitions from one state to another. The second one is that it is possible to use the same function $f$ for all transitions. These two factors make it possible to learn a single model that operates across all time steps and sequence lengths that takes advantage of shared parameters across all sequence.

In a RNN forward step, where each time step, from $t = 1$ to $t = \tau$, equations 2.27 are applied, where it is used an hyperbolic tangent as activation function and an $g$ represents an output unit. The parameters are the bias terms, $b$ and $c$, and the matrices $U$, $V$ and $W$, that are the the input-to-hidden, hidden-to-hidden and hidden-to-output connections, respectively.

$\begin{align*}
  a^{(t)} &= b + Wh^{(t-1)} + Ux^{(t)} \\
  h^{(t)} &= \tanh(a^{(t)}) \\
  o^{(t)} &= c + Vh^{(t)} \\
  y^{(t)} &= g(o^{(t)})
\end{align*} \quad (2.27)$

There is a wide variety of RNNs relationships that is possible to create with the idea of graph unrolling and parameter sharing across different time steps. It is possible to create a RNN with an one-to-one, one-to-many, many-to-one or many-to-many relation. Those relationships are show in Figure 2.10.

### 2.4.1 BackPropagation Through Time

Computing the gradient through a RNN is the same as applying the generalized backpropagation algorithm to the unrolled computational graph. Gradients obtained, by backpropagation, may then be used with any general purpose gradient-based technique to train the RNN parameters.

The total loss for a given sequence $X$, paired with a sequence of values $Y$, is the sum of the losses over all time steps, as shown in equation 2.28.
L = \sum_t L^{(t)} \hspace{1cm} (2.28)

For example, if the objective is calculate the loss function of \( y^{(t)} \) given all previous time steps of a sequence \( X = (x^{(1)}, ..., x^{(t)}) \), using as loss function the negative log-likelihood and as output unit the softmax function, then the loss function, \( L^{(t)} \) is given by equation 2.29.

\[
L^{(t)} = -\log p_{\text{model}}(y^{(t)}|x^{(1)}, ..., x^{(t)}) \hspace{1cm} (2.29)
\]

The total loss, \( L \), is calculated by equation 2.30, where \( p_{\text{model}}(y^{(t)}|x^{(1)}, ..., x^{(t)}) \) is computed with entry \( y^{(t)} \) from the output vector \( y^{(t)} \).

\[
L(\{x^{(1)}, ..., x^{(t)}\}, \{y^{(1)}, ..., y^{(t)}\}) = - \sum_t \log p_{\text{model}}(y^{(t)}|x^{(1)}, ..., x^{(t)}) \hspace{1cm} (2.30)
\]

The steps to calculate the gradient comprise the forward and backpropagation step through the computational graph. The complexity of this operation is thereby, \( O(\tau) \), and cannot be reduced by parallelization since this is an inherently sequential process, where each time step \( t \) depends on all previous time steps. This backpropagation step in a recurrent graph is known as the BackPropagation Through Time (BPTT).

For a better understanding of BPTT it will be exemplified how to calculate this task for a RNN. The learning parameters are the matrices \( U \), \( V \) and \( W \) and the bias terms \( b \) and \( c \). The different nodes of the computational graph will be indexed, \( x^{(t)}, h^{(t)}, o^{(t)} \) and \( L^{(t)} \), for each time step \( t \). For each individual node a gradient \( \nabla_N L \) is calculated recursively, for the node \( N \).

The outputs \( o^{(t)} \) are used to compute a vector of probabilities \( y \), using the softmax function. The loss used is the negative log-likelihood, as in equation 2.29. The gradient of the internal nodes from the computational graph must be calculated, in order to update the parameters. To achieve that the algorithm starts by calculating the gradient from the nodes that precedes the final loss (equation 2.31).

\[
\frac{\partial L}{\partial L^{(t)}} = 1 \hspace{1cm} (2.31)
\]
Figure 2.11: BPTT computational graph example. Red arrows represent the gradient flow from the outputs through the hidden states.

The gradient from the outputs at time step \( t \), for all outputs \( i \) is given by equation 2.32.

\[
\nabla_{o^{(t)}} L_i = \frac{\partial L}{\partial o^{(t)}_i} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o^{(t)}_i} = \hat{y}^{(t)}_i - 1_i, y^{(t)}_i
\]  \hspace{1cm} (2.32)

For the hidden state the gradient from the time step \( t = \tau \) until \( t = 1 \) must be calculated. The gradient is given by equation 2.33. This is a recurrent operation, since it needs to be computed the same number of time as the steps considered. An example is shown in Figure 2.11.

\[
\nabla_{h^{(t)}} L = (\frac{\partial h^{(t+1)}}{\partial h^{(t)}})^T \nabla_{h^{(t+1)}} L + (\frac{\partial o^{(t)}}{\partial h^{(t)}})^T \nabla_{o^{(t)}} L
\]  \hspace{1cm} (2.33)

Now the parameters are updated using the equations 2.34, for the parameters \( c, b, V, W \) and \( U \), respectively, where \( \nabla_{W^{(t)}} \) and \( \nabla_{U^{(t)}} \) are used to denote the contribution of the weights \( W \) and \( U \) at time step \( t \) to the gradient.

\[
\nabla_c L = \sum_t (\frac{\partial o^{(t)}}{\partial c})^T \nabla_{o^{(t)}} L
\]

\[
\nabla_b L = \sum_t (\frac{\partial h^{(t)}}{\partial b^{(t)}})^T \nabla_{h^{(t)}} L
\]

\[
\nabla_V L = \sum_t \sum_i (\frac{\partial L}{\partial o^{(t)}_i}) \nabla_{V o^{(t)}_i}
\]

\[
\nabla_W L = \sum_t \sum_i (\frac{\partial L}{\partial h^{(t)}_i}) \nabla_{W^{(t)} h^{(t)}_i}
\]

\[
\nabla_U L = \sum_t \sum_i (\frac{\partial L}{\partial h^{(t)}_i}) \nabla_{U^{(t)} h^{(t)}_i}
\]  \hspace{1cm} (2.34)
Analyzing the expressions it is possible to identify one of the major problems with BPTT, which is that gradients propagated over many stages tend to either vanish or explode, due to the recurrent nature of the operations, being a challenge to the network to learn long sequences relationships. Even if we assume that the parameters are such that the recurrent network is stable (can retain information, with gradients not exploding), the difficulty with long-term dependencies arises from the exponentially smaller weights given to long-term interactions (involving the sequentially multiplication of many Jacobians) compared with short-term ones. This problem is particular to RNNs. In practice, the experiments shows that if the span of the dependencies that need to be captured are increased, gradient-based optimization becomes increasingly difficult, with the probability of successful training of a traditional RNN via gradient descent rapidly reaches 0 for sequences of only a length of 10 or 20 [33].

2.4.2 Long-Short Term Memory

Long-Short Term Memory (LSTM) were introduced in 1997 [34], and were especially designed to deal with long term dependencies in recurrent architectures. The main idea is the introduction of self-loops to produce paths where the gradient can flow and making the weights on this self-loop conditioned on the context, rather than fixed. The core idea behind behind LSTMs is the the introduction of the cell state \( C_t \), where the LSTM has the ability to add or remove information to it, functioning as a memory. This process is controlled by gates mechanisms. LSTMs have three of these units, forget, external input and output gates.

**Forget Gate**

This gate has the task to decide what information it should be thrown away, or in other words, to be forgotten, from the cell state. We need to thought that the cell state represents our knowledge that we have from all the previous time steps, and depending on the task, when dealing with new information we might want to forget old information or keep it.

\[
f^{(t)} = \sigma(W_f \times [h^{(t-1)}, x^{(t)}) + b_f)
\]  

**Input Gate**

This gate has the objective to choose which information are going to be stored in the cell state. It is divided in two parts, the first one, called input gate layer, is a sigmoid layer that decides which values will update the cell state. The second is a hyperbolic tangent layer that creates a vector of new candidates to feed the cell state. This part is exactly the same as a shallow RNN, since it is making a transformation to an input \( x^{(t)} \) and the old state \( h^{(t-1)} \). The key objective is to give the network the ability to choose which information is relevant to store the cell state.
\[
i^{(t)} = \sigma(W_i \times [h^{(t-1)}, x^{(t)}] + b_i) \\
\hat{C}^{(t)} = \tanh(W_C \times [h^{(t-1)}, x^{(t)}] + b_C)
\]

(2.36)

**Update Cell State**

The cell state will be updated according to the previous steps that already decided what to do. It will multiply the cell state by the output of the forgetting gate and then add new information that was decided to update the cell state.

\[
C^{(t)} = f^{(t)} \otimes C^{(t-1)} + i^{(t)} \otimes \hat{C}^{(t)}
\]

(2.37)

**Output Gate**

The output gate has the objective of controlling what it will be passed for the next time step through the hidden state \(h^{(t)}\). This depends on the current cell state but will be a filtered version. This filtered version is computed by a sigmoid layer that decides what parts of the cell state are going to be passed for the output and by a hyperbolic tangent layer that will squish the cell state to values between 1 and -1.

\[
o^{(t)} = \sigma(W_o \times [h^{(t-1)}, x^{(t)}] + b_o) \\
h^{(t)} = o^{(t)} \otimes \tanh(C^{(t)})
\]

(2.38)

Figure 2.12: Recurrent neural networks architectures variants.

**Variants**

From LSTM similar ideas were developed, being the most known one the Gated Recurrent Unit (GRU) cell. GRU combines forget and input gates into a single gate. The cell and hidden state are also merged. This is a simpler model than LSTM but has been achieving good results as well. Figure 2.12 shows LSTM and GRU cell architectures. Several new cell architectures have been introduced. Cells like
Lattice Recurrent Unit [35], Statistical Recurrent Unit [36] or independently Recurrent Neural Network [37].

**Gradient Flow**

LSTMs cells can attenuate the vanish gradient problem that is present in RNNs, making it possible to learn longer sequences relationships. A simplified diagram of RNN and LSTM cells is shown in Figure 2.13. In the case of a LSTM, from the diagram, and omitting the hidden state \( h(t) \) from the diagram, the cell state at time step \( t \) is given by equation 2.39.

\[
C(t) = C(t-1) \otimes f + C(t-1) \otimes f_w \tag{2.39}
\]

Computing the gradient between two consecutive nodes in the computational graph is given by equation 2.40, where it is possible to see that the last three terms will tend to zero as the sequence length increases, since they are as the exactly same type of operations made in a vanilla RNN. Ignoring those terms is possible to approximate the gradient between two consecutive nodes by using equation 2.41. Figure 2.14 shows the gradient flow through the simplified diagram.

\[
\frac{\partial C(t)}{\partial C(t-1)} = f(t) + C(t-1) \frac{\partial f(t)}{\partial C(t-1)} + i(t) \frac{\partial g(t)}{\partial C(t-1)} + g(t) \frac{\partial i(t)}{\partial C(t-1)} \tag{2.40}
\]

\[
\frac{\partial C(t)}{\partial C(t-1)} = f(t) \tag{2.41}
\]

Applying BPTT in these graphs can be resumed by equation 2.42, where the forget gates keep being multiplied.

![Figure 2.13: Comparison of LSTM and a RNN structures.](image)

![Figure 2.14: BPTT graph example. Red arrows represents the gradient flow from the outputs through the hidden states.](image)
\[
\frac{\partial C(t)}{\partial C(t-x)} = \frac{\partial C(t)}{\partial C(t-1)} \frac{\partial C(t-1)}{\partial C(t-2)} \cdots \frac{\partial C(t-(x+1))}{\partial C(t-x)} = f(t) \ast f(t-1) \ast \ldots \ast f(t-x)
\] (2.42)

This attenuates the problem of vanish gradient but does not solve it fully, since the gradient still involves a chaining of multiplications in the internal nodes. If in one of those steps the forget gate is approximately zero, the whole gradient will vanish, which means that from there no information will be able to flow. This can be seen as something expected because when the forget gate is closely to zero, it means that the network is trying to forget that information, because it is not relevant for the task.
Deep Learning Models for Time-Series

Deep learning has several applications and is under intense development in the last years. Forecasting and predictive maintenance are two examples which is possible to use deep learning algorithms. Both problems are sequential tasks, where the concept of time and ordering is present and therefore, both can be modeled by a sequential learning algorithm. The default option in deep learning for sequential modeling is to use recurrent-based algorithms. Despite the recent developments in the last years, using deep learning algorithms to sequential problems still remains a challenging problem, due to vanish gradient problems that makes learning long term dependencies difficult and expensive.

Recurrent networks were specially designed to deal with sequential problems, and are the default option for problems of this nature. RNNs are capable of encoding the sequential context in their hidden state $h(t)$, which, theoretically, can capture infinitely long-term dependencies. In practice, RNNs have shown problems to capture long-term dependencies, mostly due to the vanish gradient problem. To overcome that difficult LSTM, and other variants, like GRU, have been being introduced and got an immense popularity due to the success in applications such as, language modeling, machine translation, speech recognition and others, even though the vanish gradient problem is not fully resolved. Since there is a big flexibility in the design of RNN architectures, there are many other models and training strategies that have been introduced and are being explored. Several empirical studies were developed to explore the performance of different architectures on various sequential tasks, but in [38] it was conclude that, if there were "architectures much better than the LSTM", they were "not trivial to find".

CNNs are mostly known for their applications on tasks involving images, like image classification or face recognition. But CNNs are also being applied to sequential problems, in tasks like, speech recognition [39], sentence classification [40] or audio synthesis [41]. The results obtained with CNNs are notable and promising. In [42] the authors showed that convolutional architectures can outperform recurrent architectures in several sequential modeling tasks and should be considered as a default option to these type of problems.
3.1 Recurrent Architectures

3.1.1 Recurrent Neural Network Architecture

When dealing with sequential problems we are interested in predicting the next value of a sequence conditioned on all previous time steps. This can be modeled by maximizing the likelihood of our target value given all previous steps. Given a predictor with parameters $\theta$, the task is to maximize $y(t)$ conditioned on previous steps, as shown in equation 3.1.

$$p(y|x) = \prod_{t=0}^{T} p(y(t)|x^{(0)}, ..., x^{(T)}, \theta)$$

(3.1)

RNNs architectures are specially designed to deal with sequential problems. When a RNN is used to deal with sequential information the network is trained to condition each output $y(t)$ based on previous sequence of inputs $(x^{(0)}, ..., x^{(t)})$. To encode all previous information the RNN architecture uses a hidden state $h$, as in equation 3.2, where $h^{(t-1)}$ encodes all information up until $t - 1$, and the output $y$ is a function of $h^{(t-1)}$ and the input $x^{(t)}$. The network parameters are represented by $\theta$.

$$y(t) = f(h^{(t-1)}, x^{(t)}; \theta)$$

(3.2)

This characteristics makes RNNs architectures a natural solution to learn the probability distribution function introduced in equation 3.1. Depending on the RNN cell used, the function $f$ applied in equation 3.2 is different. A shallow RNN, GRU and LSTM cells will be implemented and compared in the experiments. An example of the application of equation 3.2 is shown in Figure 3.1.

![Figure 3.1: Unfolded recurrent architecture.](image)

3.1.2 Encoder-Decoder

The Encoder-Decoder RNN architecture was introduced in [24], and it combines two different RNNs, one for the encoder and other for the decoder. The first one has the task of encoding a sequence with an arbitrary length sequence to a fixed vector length, whereas the second RNN is used to decode the information encoded in that vector into another sequence. An example is given in Figure 3.2.

![Figure 3.2: Encoder-Decoder architecture.](image)

From a probabilistic perspective, this method learns a probability distribution of a future sequence based on previous values [24], as in equation 3.3, where $T^1$ and $T^2$ represent different sequence lengths.
for the encoder and decoder. This differs from RNNs architectures, because it was developed to predict several steps ahead, instead of the next one, as in equation 3.1.

\[
p(y^{(1)}, ..., y^{(T_1)}) = \prod_{t=0}^{T_1} p(y^{(t)}|x^{(0)}, ..., x^{(T_2)}, y^{(0)}, ..., y^{(t-1)}, \theta) \quad (3.3)
\]

The model will encode all information about the input sequence in the hidden state, \( h^{(t)} \), by applying equation 2.25. The hidden state is a latent representation from the input sequence \( (x^{(0)}, ..., x^{(T_2)}) \), and is usually referred as the context vector, \( C \). The decoder is trained to predict the next value \( y^{(t)} \), given the previous hidden state \( h^{(t-1)} \) and also conditioned on the context vector \( C \), as in 3.4, where the function \( f \) applied depends on the cell state used, as in RNNs architectures.

\[
y^{(t)} = f(h^{(t-1)}, y^{(t-1)}, C; \theta) \quad (3.4)
\]

**Attention System**

A new approach to encoder-decoder models, introduced in [22], exploits the bottleneck of encoding an arbitrary sequence length into a fixed length vector. The objective is to allow the decoder learn which is the hidden state (or context vector), from the encoder that is most valuable to predict the next output. Doing this, enables each output of the decoder to choose the context vector \( C \), based on the set of positions where the input sequence carries more information.
Applying this attention system, the output from the decoder is given by 3.5, where the difference, compared with the encoder-decoder model without attention, is that the output $y^{(t)}$ is now conditioned on a different context vector $C_i$ for each $i$, as shown in Figure 3.3

$$y^{(t)} = f(h^{(t-1)}, y^{(t-1)}, C_i; \theta)$$ (3.5)

The context vector will be a weighted sum of the encoder hidden states, $h^{(i)}$. Each $h^{(i)}$ contains information about the whole sequence until the $i$-th input, but it has a very strong focus on the $i$-th input. The context vector is then given by equation 3.6.

$$C_{(i)} = \sum_{j=1}^{T} \alpha_{ij} h_j$$ (3.6)

Each weight $\alpha_{ij}$ is calculated by equation 3.7, where $e_{ij}$ is a function of the previous hidden state of the decoder and the $j$-th hidden state of the encoder, $e_{ij} = f(h_{decoder}^{(t-1)}, h_{encoder}^{(j)})$. This architecture is resumed in Figure 3.4.

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T} \exp(e_{ik})}$$ (3.7)

![Figure 3.4: Attention Layer.](image)

### 3.1.3 Dilated Recurrent Neural Network

Using RNN architectures to train on long sequences results in three major problems: complex dependencies, gradient problems (vanish/exploding) and long training times. Dilated Recurrent Neural Networks (DilatedRNN) were introduced to deal with these limitations [43]. They are characterized by a multi-resolution dilated recurrent skip connections which can decrease the number of parameters necessary in tasks involving long term dependencies.

DilatedRNN are similar to dilated convolution models [41], but under a recurrent setting. The main idea is to use dilated recurrent skip connections to alleviate the gradient problems and enable the training of longer sequences. The use of dilated recurrent connections can also reduce the number of
parameters, increasing the computational efficiency. One main idea is that by stacking several layers the DilatedRNN model is able to learn temporal dependencies at different time scales. The DilatedRNN model is shown in Figure 3.5.

![DilatedRNN model with 3 layers.](image)

Figure 3.5: DilatedRNN model with 3 layers.

On a vanilla recurrent connection the hidden state $h(t)$ is a function of the previous hidden state $h(t-1)$ and the current input $x(t)$, as in equation 3.2. When a dilated recurrent skip connections is used the hidden state $h(t)$ is a function of the current input $x(t)$ and the hidden state $h(t-s)$, as shown in equation 3.8, where $h(t)$ is the hidden state at time step $t$ and $s$ is the skip length or the dilation of the $l$-th layer. The function $f$ depends on the recurrent cell used.

$$h_i^{(t)} = f(x_i^{(t)}, h_i^{(t-s)})$$

Dilated recurrent skip connections allow information to travel along fewer edges and can be calculated in a parallel fashion, thus increasing computational efficiency. This can be seen in Figure 3.6, where the input is processed in parallel instead of a unique sequence. In [41] the authors used exponentially increasing of dilation factors, to extract complex time dependencies.

![Dilated connections.](image)

![Dilated layer with optimized inputs.](image)

Figure 3.6: Dilated recurrent skip connections can reduce the number of parameters used and allow parallel computations.
3.2 Convolutional Architectures

3.2.1 Wavenet

Introduced as a generative model for raw audio, [41], the Wavenet model is an autoregressive probabilistic model, where the probability of each sample is conditioned on all previous ones.

Wavenet is a convolutional neural network architecture, that uses stacked convolution layers to model the conditional probability distribution (equation 3.1). Causal convolutions are used to deal with the time ordering of samples and guarantee that the model at timestep \( t \) cannot depend on any future values \( x(t+1) \). An example of a causal convolution is depicted in Figure 3.7.

![Causal convolution](image)

Using large receptive fields size is preferred to increase the amount of information from past samples. The receptive field size increases linearly with the number of layer in the network if simple causal convolutions are employed (Figure 3.7). To increase the receptive field size without greatly increasing the computational cost, dilated causal convolutions may be used. Dilated convolution (also called convolution with holes) is a convolution where the filter has a larger area than its length by skipping inputs, with a length that is equivalent to the dilation factor. This allows the model to have a larger receptive field size, using the same number of layers. A comparison between causal and dilated causal convolutions is given in Figure 3.8. This enables the network to have a receptive field size that increases exponentially with the depth of the model.

![Comparison of receptive field size](image)

Wavenet model uses residual and parameterised skip connections, to speed up convergence. In
residual connections, the output of a layer is added with the input of that layer, as demonstrated in equation 3.9, which has been shown to benefit very deep networks [44].

\[
\text{output} = \text{activation}(x + f(x))
\]  

(3.9)

\[
Z = \tanh(W_{f,k} \ast x) \otimes \sigma(W_{g,k} \ast x)
\]  

(3.10)

Gated activation units are used, as in equation 3.10, where \( k \) is the layer index, \( f \) and \( g \) the filter and gate, respectively and \( W \) represents the convolution filter that is learned during training. The Wavenet model architecture can be seen in Figure 3.9.

Figure 3.9: Wavenet model architecture.

### 3.2.2 Temporal Convolution Block

The work introduced in [42], showed a simple convolution neural network architecture outperforming several recurrent neural architectures for sequence modeling and it was inspired in recent works about sequential data. The authors stated that convolution networks should be regarded as a natural starting point for sequential tasks. The model introduced is called Temporal Convolution Network (TCN). The model was designed to combine simplicity, autoregressive prediction and very long memory. The model architecture can be seen in Figure 3.2.2.

Similarly to the Wavenet architecture TCN uses dilated convolutions and residual connections to build very long and effective receptive fields. TCN has several advantages over RNN architectures for sequential tasks. Convolutional models can be parallelized, unlike RNNs where the output at time step \( t \) must wait for all previous time steps to be computed. They have a flexible receptive field size that can be changed in several ways, like changing the number of layers or the kernel size from dilated convolutions. Stable gradients are another advantage, because the backpropagation algorithm do not
suffer from exploding/vanish gradients like in RNNs due the recurrent connections. Low memory for training and a variable length inputs are also two advantages. A drawback is the memory storage needed during evaluation, since all sequence must be kept in memory, which is a problem that RNN don’t have because they only need to store the last hidden state to compute the next output.

### 3.2.3 Quasi Recurrent Neural Network

The Quasi Recurrent Neural Network (QRNN) [45] was developed to avoid some limitations that are inherent to RNN architectures, that limits the parallelism and make RNNs models hard to train on long sequences. QRNN alternates convolutional layers with a recurrent pooling operation that can be applied in parallel. This can speed up the model training, when compared to RNNs.

The two main components of a QRNN are the convolution layer and the pooling operation layer. The first component applies a convolution along time dimension, where giving an input $X$ of $Tn$-dimensional vectors $(x_1, \ldots, x_T)$, it will be produced three sequences of $m$-dimensional vectors (see equation 3.11), where $m$ is the number of filters used. Causal convolutions must be used to avoid any time leakage.
\[
Z = \tanh(W_z * X)
\]
\[
F = \sigma(W_f * X)
\]
\[
O = \sigma(W_o * X)
\]  

(3.11)

The vector \( Z \), which represents the candidate vectors, is passed through a \( \tanh \) non-linearity. Vectors \( F \) and \( O \), which will be used in the elementwise gates operation in the pooling layer, are squashed by a sigmoid layer. The weight matrices \( W_z \), \( W_f \) and \( W_o \in \mathbb{R}^{k \times n \times m} \) are the parameters to be learned, where \( k \) is the filter size, \( n \) the number of input vectors and \( m \) the number of filters used.

The recurrent pooling layer operations are inspired in the element-wise operations used in LSTM cells. The objective is to have a function controlled by gates that are able to mix states across timesteps that act independently on each channel of the state vector. In the paper 3 types of pooling operations are introduced. \( f \)-pooling (equation 3.12), \( f o \)-pooling (equation 3.13) and \( if o \)-pooling (equation 3.14).

\[
h^{(t)} = f^{(t)} \otimes h^{(t-1)} + (1 - f^{(t)}) \otimes z^{(t)}
\]  

(3.12)

\[
c^{(t)} = f^{(t)} \otimes c^{(t-1)} + (1 - f^{(t)}) \otimes z^{(t)}
\]
\[
h^{(t)} = o^{(t)} \otimes c^{(t)}
\]  

(3.13)

\[
c^{(t)} = f^{(t)} \otimes c^{(t-1)} + i^{(t)} \otimes z^{(t)}
\]
\[
h^{(t)} = o^{(t)} \otimes c^{(t)}
\]  

(3.14)

A single QRNN layer applies an input-dependent pooling, followed by a gated linear combination of convolution features. An example of a QRNN layer is given in Figure 3.11.

![Figure 3.11: Quasi Recurrent Neural Network layer.](image)
Chapter 4

Methodology

As already stated in previous chapters, time series problems appear in different fields and have a broad range of applications. The performance and versatility of the models presented in Chapter 3, will be tested on two different time-series problems, with the objective of understanding which are the models’ characteristics that yield better results and are worth to be further investigated.

The first is wind power produced in Portugal between 2010 and 2016. The second one is related to predictive maintenance and the goal is to predict in advance when turbofan engines will fail.

In the following sections, a detailed explanation will be given for the two datasets, where it will also be presented the methodology used on both problems to compare the models. The methodology is the same for all algorithms to have a fair and correct comparison among them. Before that, some practices used on both problems will be explained. In appendix A, different diagrams are exposed to resume the methodology used, as well the training strategy applied to train the models on both datasets.

All experiments were developed in Python and the deep learning models were developed with PyTorch [46].

4.1 Data Normalization

Data normalization or feature scaling is a preprocessing step that is used depending on the machine learning algorithm used. It is known to be beneficial to some methods that are comparing features with different scales. In theory, feature scaling is not necessary to apply when it is used neural networks, because the model should be able to account for the differences in scale. Nonetheless, scaling and centering all features around the origin allows neural network’s hidden units to have a smaller chance of getting saturated, improving the convergence speed. It is also important to equally weigh each feature during training, to improve the optimization step.

When applied to neural networks, there is no obvious answer to which normalization should be used. One of the common ones is the standardization, shown in equation 4.1, which makes each feature from the training data to have zero mean and unit variance, where \( \mu \) and \( \sigma \) are the mean and variance, respectively.

\[
\text{Table 4.1.}
\]
\[ X_{\text{normalized}} = \frac{X - \mu_{\text{data}}}{\sigma_{\text{data}}} \tag{4.1} \]

The Min-Max scaling is also frequently used to transform the data to be in the interval \([-1, 1]\).

\[ X_{\text{normalized}} = \frac{X - X_{\text{min}}}{X_{\text{max}} - X_{\text{min}}} \tag{4.2} \]

## 4.2 Cross-validation

The main objective when training a machine learning algorithm is its generalization, i.e. how it performs on unseen data. The concepts of under and over-fitting may occur when a model is trained. The former happens when the model is not capable of modeling the data, leading to a poor performance on the test set. Latter occurs when the algorithm memorize the training data, this often arises due to models that are over-parameterized and complex. This also makes the model to perform poorly in the test set, since the model has memorized the training data instead of learning which is the underlying probability distribution. Our objective is to find a model that is able to avoid both cases, meaning that, is able to generalize well to unseen data. Those concepts are shown in Figure 4.1.

![Figure 4.1](image1.png)

(a) Under Fitting.  (b) Over fitting.  (c) Good fitting.

Figure 4.1: Generalization example. It is possible to see that a good fitting to the training data does not translate a good generalization.

The cross-validation (CV) technique will be used to control the over-fitting problem. This procedure randomly splits the data set in \(K\) mutually exclusive folds. The model is then trained on \(K - 1\) folds and tested in the one that was left out. The procedure is then repeated \(K\) times. This method is known as \(K\)-fold CV. The "best" model will be the one with lowest mean prediction error across all steps. This is represented in Figure 4.2.

![Figure 4.2](image2.png)

Figure 4.2: Example of a 5-fold CV strategy.

Additional care should be taken when using CV technique to time-series problems due to their inher-
ent autocorrelation (time ordering) characteristic. If the standard \( K \)-fold technique would be applied to a time-series problem it would naively ignore the sequential nature of time. To account for this problem a different procedure is used. An example is seen in Figure 4.3, where the machine learning model will always train on past values to predict future values, guaranteeing the time ordering.

![Figure 4.3: Example of a sliding window CV procedure for sequential data.](image)

### 4.3 Hyperparameter Optimization

A model hyperparameter is a configuration that is external to the model and whose value cannot be estimated from the data. These are used in the process of estimating the model parameters (weights and biases) and need to be specified beforehand. The number of layers, hidden nodes or the learning rate are some examples of a hyperparameter. Unfortunately, the hyperparameters set chosen has a big influence on the results obtained and finding the best combination of hyperparameter can be an expensive procedure when it is made by trial and error. Techniques like grid search or random search are usually applied. However these two methods are inefficient and computationally expensive, since the hyperparameter space is usually vast and of high dimensionality. To ease this challenge techniques like Bayesian optimization with Gaussian priors may be employed. A detailed explanation of the method and the advantages is given in [47]. The python package Scikit-Optimize[48] was used.

### 4.4 Datasets

#### 4.4.1 Wind Power Generation

Power forecasting is of tremendous importance for electrical utilities and network operators to manage the power system operation. At any moment, the balance between consumed and generated power must be maintained, otherwise the grid's stability and power demand cannot be satisfied, if power consumption is higher than production. Thus, it is of big importance to predict in advance what will be the power generated by renewable sources to prepare the system dispatch in advance. Making a dispatch for the wind power generation is no easy task, since it is a very volatile resource, because it is a direct consequence of the wind speed. An example is shown in Figure 4.4, where it is possible to see that wind generation is highly erratic.

The data was collected by Redes Energéticas Nacionais (REN) and consists on the injected wind power in the Portuguese power system. It was sampled at a 15 minutes resolution from the first day.
of 2010 until the last day of 2016. The data collected pertains to all wind farms that are connected to REN’s telemetry system. This information can be publicly accessed at [49].

Figure 4.4: Wind power generated in Portugal during January 2011.

Usually, the time horizon to predict the power generation is variable and depends on its purpose. Different time horizons can be seen in Table 4.1, accordingly to [50].

Table 4.1: Forecasting time horizons.

<table>
<thead>
<tr>
<th>Time Horizon</th>
<th>Range</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Short-Term</td>
<td>Few seconds</td>
<td>- Electricity Market Clearing</td>
</tr>
<tr>
<td></td>
<td>30 minutes</td>
<td>- Regulation Actions</td>
</tr>
<tr>
<td>Short-Term</td>
<td>30 minutes</td>
<td>- Economic Load Dispatch Planning</td>
</tr>
<tr>
<td></td>
<td>6 hours</td>
<td>- Load Increment/Decrement Decisions</td>
</tr>
<tr>
<td>Medium-Term</td>
<td>6 hours</td>
<td>- Generator Online/Offline Decisions</td>
</tr>
<tr>
<td></td>
<td>1 day</td>
<td>- Operational Security in Day-Ahead Electricity Market</td>
</tr>
<tr>
<td>Long-Term</td>
<td>1 day</td>
<td>- Unit Commitment Decisions</td>
</tr>
<tr>
<td></td>
<td>1 week</td>
<td>- Reserve Requirement Decisions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Maintenance Scheduling to Obtain Optimal Operating Cost</td>
</tr>
</tbody>
</table>

Several methods have been developed to tackle this problem, where the approach chosen is highly dependent on the time horizon that we want to predict. The most common methods reported in the literature can be divided into the following classes:

- **Persistence method** - This is the simplest method used and is known as the “naive predictor”. This method assumes that the wind power generated at time $t + 1$ will be equal to the one at time $t$.

- **Physical approach** - This methods uses information about the atmosphere physical behavior and tries to parameterize the onsite conditions of the wind farm with the weather conditions, extrapolating from that the wind power generation for those conditions. These methods use complex mathematical models that are computationally expensive.
• Statistical approach - This method uses the observed data points to model the wind power generation, where there is not any domain knowledge introduced with a physical model, being only based on patterns that arises from the data. Several methods can be used to predict these patterns. The most common approaches are based on time-series and artificial neural networks methods. On the time-series methods, ARMA models, and their variations, like ARIMA, SARIMA or ARIMAX, are usually the ones used. Linear predictions or exponential smoothing are examples of other time-series based models. Neural networks based methods like MLP, RNN, CNN, Radial Basis Function (RBF), have also been used.

• Hybrid methods - This approach tries to benefit from both statistical and physical approaches or short and long term forecasting methods.

The approach used falls in to the category of statistical methods. Short-term and medium-term time horizon will be predicted. More specifically, time horizons of one, six and 24 hours ahead will be forecast. This means that three different experiments will be made, where depending on the horizon a different number of predictions will be made. For example, in an one hour horizon forecast we will make four predictions, for 15, 30, 45 and 60 minutes ahead. This will be made consecutively, this means that we will use the 15 minutes ahead prediction as input to predict the next point, and so on. The training data consists only on the past measurements values of the wind generated power and range from the first day of 2010 until the last day of 2015, while the full year of 2016 will be predicted. In the next sections, the methodology to test the different architectures is going to be presented. For a current status on wind power forecasting see [50] and [51].

Data preparation

To use the algorithms from Chapter 3, we should make a data transformation to take advantage from the recurrent nature of the data. The data collected can be represented in a matrix form such as, $X = [x^{(1)}, x^{(2)}, ..., x^{(T)}]$, where each $x^{(t)} \in \mathbb{R}$ represents the wind power generated at time-step $t$. The data will be transformed to $X = [x_1, x_2, ..., x_{T-p}]^T$, where each $x_t \in \mathbb{R}^{1 \times p}$, represents an ordered sequence of wind power generated of length $p$. For each training step the label will also be an ordered sequence, and depending on the model used, can be represented as $Y = [y_1, y_2, ..., y_{T-n}]^T$, where each $y_t \in \mathbb{R}^{1 \times n}$. This methodology is shown in Figure 4.5.

![Data preprocessing example](image)

**Figure 4.5:** Data preprocessing example for a training sequence of length $p$ and a training label of length $n$.

When the model has an encoder-decoder architecture, then the transformation applied is the one in
Figure 4.5, because this model is used to maximize $p(x_{t+1}, \ldots, x_{t+n} | x_t, \ldots, x_{t-p})$. The remaining models were conceived to maximize $p(x_{t+1} | x_t, x_{t-1}, \ldots, x_{t-p})$, and thus the label is different. Because of that the transformation is also different. In Figure 4.6 an example of this transformation can be seen. The window length, $p$, that represents the number of steps to use in the training phase, will be an hyperparameter that will be optimized during the training phase. The data will be normalized using standardization (zero mean and nit variance), as in equation 4.1.

![Figure 4.6: Data preprocessing example for a training sequence of length $p$. On this type of models the input and output have the same length, being the output equal to the input shifted one time-step.](image)

**Prediction Step**

Depending on the model used, the procedure to obtain predictions are different. When encoder-decoder models are used the output will already be of the desired length.

![Figure 4.7: Example of a model using its own predictions as input enabling the prediction of several steps ahead.](image)

The remaining models are trained to predict $x^{(t+1)}$, but our objective is to predict more than one point ahead. To achieve this, in the prediction step, the model will use its own predictions of $\hat{x}^{(t+1)}$ to predict $\hat{x}^{(t+2)}$. This is possible to do sequentially until we achieve the number of time steps that are desired. This process is shown in Figure 4.7.

**Metrics**

Three metrics will be used to evaluate and compare the performance of the different algorithms used in this dataset. The Mean Squared Error (MSE) and the Mean Absolute Error (MAE), as in equations 4.3
and 4.4, respectively, where \( y_{\text{true}} \) is the true value of wind power, \( y_{\text{predicted}} \) is the wind power prediction and \( N \) is the total number of time-steps predicted. The Mean Absolute Percentage Error (MAPE) it is also used and it represents the average difference between the actual and the forecast expressed as a percentage of the true value, as in equation 4.5.

\[
MSE = \frac{\sum_{i=1}^{N} (y_{\text{true}}^i - y_{\text{predicted}}^i)^2}{N} \tag{4.3}
\]

\[
MAE = \frac{\sum_{i=1}^{N} \|y_{\text{true}}^i - y_{\text{predicted}}^i\|}{N} \tag{4.4}
\]

\[
MAPE = \frac{100}{N} \times \frac{\sum_{i=1}^{N} \|y_{\text{true}}^i - y_{\text{predicted}}^i\|}{y_{\text{true}}^i} \tag{4.5}
\]

### 4.4.2 C-MAPSS Turbofan

Commercial Modular Aero-Propulsion System Simulation (C-MAPSS) is a turbofan simulation model described in [52]. C-MAPSS was used to generate a simulated run-to-failure dataset from a turbofan engine and it is published in NASA’s prognostics center of excellence repository, that is publicly available in [53]. A big bottleneck in data driven approaches to tackle problem in predictive maintenance scope is the lack of availability run-to-failure datasets. This simulated dataset allows researchers to build, test and benchmark different approaches to this problem. An extensive analysis and performance benchmark for this dataset is given in [54].

<table>
<thead>
<tr>
<th>Data set</th>
<th>FD001</th>
<th>FD002</th>
<th>FD003</th>
<th>FD004</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train Units</td>
<td>100</td>
<td>260</td>
<td>100</td>
<td>249</td>
</tr>
<tr>
<td>Test Units</td>
<td>100</td>
<td>259</td>
<td>100</td>
<td>248</td>
</tr>
<tr>
<td>Operation Conditions</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>Fault Conditions</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

The C-MAPSS dataset is composed of four sub-datasets with different operation and fault conditions. Each sub-dataset is further divided in training and test sets, as represented in Table 4.2. Within each dataset, each engine is considered from a fleet of engines of the same type and where each time series is from a single engine. Each engine starts with different levels of initial wear and manufacturing variation which is unknown. These wear and variations are considered as a normal behavior for each engine. The engine is operating normally at the start of each time series, and develops a fault at some point. In the training set, the fault grows in magnitude until engine failure. In the test set, the time series ends some time prior to the failure. The objective is to predict the number of remaining operational cycles before failure in the test set, i.e., the number of operational cycles left in the engine. This is called the Remaining Useful Life (RUL). The data consists of multiple multivariate time series measurements. More specifically, there are 21 sensors measurements and three operational settings. It is not known
what each sensor is measuring. Each row in the data is a snapshot taken during a single operation time cycle, for a certain engine. An example of three sensor measurements is given in Figure 4.8.

Figure 4.8: Example from 3 sensor measurements of dataset FD001. Gray color represents raw data and colored lines are rolling mean values.

Several methods can be used to tackle this problem [54]. One can use a functional mapping between a health index from the engines to predict the RUL or apply a similarity-based matching. The strategy used in this work is to use a functional mapping between sensor measurements to predict the RUL.

Data Preparation

Similarly to the wind dataset, the data must be transformed. Since it is known that there are six operation conditions, a clustering algorithm was applied to obtain those conditions in an unsupervised way. The three sensors that measure the operational setting were used in the clustering. The assigned operation condition is then used as a feature to our problem. HDBScan algorithm was used [55]. The Min-Max normalization was used, where each sensor was normalized taking in account the operational setting assigned, as in equation 4.6, where c and i are the operation condition and the sensor number, respectively.

\[
X_{c,i}^{\text{normalized}} = \frac{X_{c,i} - X_{\text{min}}^{c,i}}{X_{\text{max}}^{c,i} - X_{\text{min}}^{c,i}}, \forall c, i
\]  

(4.6)

Our target is to predict the RUL for each engine, which decreases linearly with the number of cycles. This means that the system is degrading linearly along with time, which, in practical situations, is not true. Usually, the system degradation is negligible at the beginning and it starts to deteriorate after the occurrence of some failure. To account for this, a piece-wise RUL label is used where the limit is settled at 130 time cycles similarly to the approach used in [56]. An example is showed in Figure 4.9.

Each engine can be represented in matrix form as \(X_n = [x_1, x_2, ..., x_p] \in \mathbb{R}^{P \times T_n}\) where \(x_p = [x_p^{(1)}, x_p^{(2)}, ..., x_p^{(T)}] \in \mathbb{R}^{1 \times T_n}\) is the sensor measurement \(p\) for the life time of engine \(n\). The data is transformed to look like \(X = [U_1, U_2, ..., U_K]^{T} \in \mathbb{R}^{K \times T_n \times P}\) where \(n\) is the sequence length and \(U_i = [x_i^{(1)}, x_i^{(2)}, ..., x_i^{(n)}]^{T} \in \mathbb{R}^{n \times P}\) is a matrix representing the all measurements until time step \(n\) for all sensors, where \(x_i^{(n)}\) is the vector representing all sensors measurements from time step \(n\). The transformation is represented in Figure 4.10. The sequence length will be optimized during training.
Three metrics will be used to evaluate and compare the performance of the different algorithms tested in this dataset. MSE and MAE and a scoring function introduced in [56], which gives different penalties when the RUL is underestimated or overestimated, and this score function account for this fact. In predictive maintenance problems, overestimating the estimated RUL of a machine has worst consequences when compared to a underestimation. The scoring function is given as in equation 4.7, where \( n \) is the total number of test units and \( h_i = RUL_{predicted} - RUL_{true} \) is the error for the test unit \( i \). From now on this metric is referred as score.

\[
S = \begin{cases} 
\sum_{i=1}^{n}(e^{-\frac{h_i}{\theta}} - 1) & \text{when, } h_i < 0 \\
\sum_{i=1}^{n}(e^{\frac{h_i}{\theta}} - 1) & \text{when, } h_i \geq 0 
\end{cases}
\] (4.7)
Chapter 5

Results

The models introduced in Chapter 3 are going to be compared in the datasets introduced in Chapter 4. The methodology used is the one presented in Chapter 4. For each dataset the results will be presented and analyzed. The hyperparameters space used in the optimization algorithm, for each model, will be shown, as well as, the learning rate, batch size, loss function and optimization algorithm used. The hyperparameter set obtained, for each model, by the hyperparameter optimization algorithm, will be shown. These sets were the ones that were used to obtain the results shown.

5.1 Wind Power Generation

The methods presented in Chapter 3 were tested in the wind power generation forecasting problem. Across the experiments it was used ADAM as optimization algorithm and MSE as loss function. The learning rate chosen was 0.001 across all experiments and the batch size used was 256. The hyperparameters space that was used in the optimization process are shown in Tables 5.1, 5.2 and 5.3. To control over-fitting a 3-fold CV with sliding window technique was used.

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of lags used</td>
</tr>
<tr>
<td>RNN Architectures</td>
<td>[50, 1000]</td>
</tr>
<tr>
<td>DRNN Architectures</td>
<td>[50, 1000]</td>
</tr>
<tr>
<td>Encoder-Decoder Architectures</td>
<td>[50, 1000]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Number of lags used</td>
</tr>
<tr>
<td>TCN</td>
<td>[50, 1000]</td>
</tr>
<tr>
<td>QRNN</td>
<td>[50, 1000]</td>
</tr>
</tbody>
</table>
RNNs are capable of learning long term dependencies and are able to generalize for longer horizons. When the time horizon is wider, we can see that the best results are obtained with convolutional based models. When the time horizon is lower, the models are able to follow the right pattern, whereas for a longer horizon the model starts to lose accuracy. This behavior is shown in Figures 5.1 and 5.2 where it is possible to see that for a lower time horizon, the models achieve better performance. This was expected since the uncertainty increases when we want to predict several steps ahead.

Persistence method, introduced in Chapter 4, was used as a baseline since it is widely used as a reference model. The results obtained in the test set (year of 2016), for one, six and 24 hour horizons, are shown in Table 5.4, where MSE and MAE are presented in MW units and MAPE is presented in percentage.

<table>
<thead>
<tr>
<th>Model</th>
<th>1 hour MSE</th>
<th>1 hour MAE</th>
<th>1 hour MAPE</th>
<th>6 hour MSE</th>
<th>6 hour MAE</th>
<th>6 hour MAPE</th>
<th>24 hour MSE</th>
<th>24 hour MAE</th>
<th>24 hour MAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>12.24 × 10^3</td>
<td>75.67</td>
<td>8.55</td>
<td>175.88 × 10^3</td>
<td>285.41</td>
<td>36.87</td>
<td>679.10 × 10^3</td>
<td>592.07</td>
<td>84.35</td>
</tr>
<tr>
<td>RNN</td>
<td>6.49 × 10^3</td>
<td>53.18</td>
<td>5.68</td>
<td>139.70 × 10^3</td>
<td>247.73</td>
<td>33.54</td>
<td>678.26 × 10^3</td>
<td>602.38</td>
<td>93.08</td>
</tr>
<tr>
<td>GRU</td>
<td>6.49 × 10^3</td>
<td>52.27</td>
<td>5.55</td>
<td>131.79 × 10^3</td>
<td>238.77</td>
<td>29.52</td>
<td>608.25 × 10^3</td>
<td>543.59</td>
<td>70.59</td>
</tr>
<tr>
<td>LSTM</td>
<td>6.33 × 10^3</td>
<td>52.83</td>
<td>5.61</td>
<td>129.84 × 10^3</td>
<td>239.97</td>
<td>33.07</td>
<td>547.24 × 10^3</td>
<td>539.32</td>
<td>89.40</td>
</tr>
<tr>
<td>DRNN - RNN</td>
<td>6.66 × 10^3</td>
<td>54.07</td>
<td>5.77</td>
<td>137.58 × 10^3</td>
<td>245.88</td>
<td>31.93</td>
<td>601.13 × 10^3</td>
<td>556.37</td>
<td>89.47</td>
</tr>
<tr>
<td>DRNN - GRU</td>
<td>6.34 × 10^3</td>
<td>52.46</td>
<td>5.70</td>
<td>131.21 × 10^3</td>
<td>243.14</td>
<td>36.87</td>
<td>547.20 × 10^3</td>
<td>559.63</td>
<td>98.92</td>
</tr>
<tr>
<td>DRNN - LSTM</td>
<td>6.37 × 10^3</td>
<td>52.72</td>
<td>5.87</td>
<td>129.64 × 10^3</td>
<td>243.89</td>
<td>38.11</td>
<td>554.35 × 10^3</td>
<td>558.98</td>
<td>103.20</td>
</tr>
<tr>
<td>Wavenet</td>
<td>6.67 × 10^3</td>
<td>54.41</td>
<td>6.06</td>
<td>179.40 × 10^3</td>
<td>283.01</td>
<td>30.14</td>
<td>537.86 × 10^3</td>
<td>547.47</td>
<td>101.97</td>
</tr>
<tr>
<td>TCN</td>
<td>6.39 × 10^3</td>
<td>52.68</td>
<td>5.68</td>
<td>125.24 × 10^3</td>
<td>234.72</td>
<td>30.18</td>
<td>544.05 × 10^3</td>
<td>526.74</td>
<td>68.57</td>
</tr>
<tr>
<td>ORNN</td>
<td>6.40 × 10^3</td>
<td>52.76</td>
<td>5.62</td>
<td>132.21 × 10^3</td>
<td>242.52</td>
<td>32.30</td>
<td>555.63 × 10^3</td>
<td>546.89</td>
<td>90.36</td>
</tr>
<tr>
<td>EncDec RNN</td>
<td>7.70 × 10^3</td>
<td>60.37</td>
<td>8.25</td>
<td>155.62 × 10^3</td>
<td>273.29</td>
<td>41.98</td>
<td>613.75 × 10^3</td>
<td>558.80</td>
<td>76.15</td>
</tr>
<tr>
<td>EncDec GRU</td>
<td>6.74 × 10^3</td>
<td>54.07</td>
<td>5.70</td>
<td>142.17 × 10^3</td>
<td>249.17</td>
<td>30.92</td>
<td>590.26 × 10^3</td>
<td>560.79</td>
<td>91.45</td>
</tr>
<tr>
<td>EncDec LSTM</td>
<td>6.94 × 10^3</td>
<td>56.71</td>
<td>6.59</td>
<td>132.51 × 10^3</td>
<td>243.57</td>
<td>32.92</td>
<td>562.07 × 10^3</td>
<td>566.40</td>
<td>109.94</td>
</tr>
<tr>
<td>Attention RNN</td>
<td>7.22 × 10^3</td>
<td>56.85</td>
<td>6.29</td>
<td>140.03 × 10^3</td>
<td>255.58</td>
<td>32.60</td>
<td>608.93 × 10^3</td>
<td>577.95</td>
<td>86.40</td>
</tr>
<tr>
<td>Attention GRU</td>
<td>6.71 × 10^3</td>
<td>54.71</td>
<td>5.96</td>
<td>139.95 × 10^3</td>
<td>247.79</td>
<td>28.37</td>
<td>544.83 × 10^3</td>
<td>538.70</td>
<td>91.64</td>
</tr>
<tr>
<td>Attention LSTM</td>
<td>7.02 × 10^3</td>
<td>56.82</td>
<td>6.70</td>
<td>142.79 × 10^3</td>
<td>254.25</td>
<td>32.60</td>
<td>571.68 × 10^3</td>
<td>539.70</td>
<td>79.28</td>
</tr>
</tbody>
</table>

Analyzing the results shown in Table 5.4 it is possible to see that all models are achieving a lower error than the baseline, especially for the one hour horizon. This means that our models are correctly identifying the short term wind patterns. The quality of our predictions decreases when we increase the time horizon. This was expected since the uncertainty becomes higher when we want to predict several steps ahead. This behavior is shown in Figures 5.1 and 5.2 where it is possible to see that for a lower horizon the models are able to follow the right pattern, whereas for a longer horizon the models start to lose accuracy.

Recurrent-based models (LSTMs and GRUs) achieved better performance for the one hour horizon. When the time horizon is wider, we can see that the best results are obtained with convolutional based methods, except for the six hour horizon in the MAPE, where the best result was achieved by the encoder-decoder model with attention and a GRU cell. This suggests that convolution based methods are capable of learning long term dependencies presented and are able to generalize for longer sequences.

As stated in Chapter 4, wind power generation forecast is directly related with wind speed, which is a very
unstable resource, meaning that using an autoregressive model to predict several steps ahead is a hard task. This can be visualized in the autocorrelation plot (see Figure 5.3), where the correlation between lags rapidly decreases. The fact that the results obtained for a longer time horizon are much worse is a consequence of this, because the correlation between the wind power generated in the last hour is stronger than the correlation for 6 or 24 hour horizons. This is an indication that wind power generation does not have a clear and predictable pattern throughout the days. The forecast accuracy could be improved if other features would be used, like temperature, humidity, pressure or wind speed, since those are related with wind power generation, thus giving valuable information to the problem [57].

In Figures 5.1 and 5.2 for one and six hour horizon all models are able to follow the global trend, but when the horizon is 24 hours the results rapidly deteriorate.
The small difference between errors across models (specially in the MSE and MAE metrics), can also be seen as a directly consequence of the lack of autocorrelation. For example, it would be expected that a recurrent network with a LSTM cell would achieve a better result than a RNN-based recurrent network, but this is not reflected in our results, where the differences between the models are slim. This is supported by the learning curves presented in Figure 5.4 where we can see that the models are converging to approximately the same training loss value. If more complex relationships would be presented in the data, the advantage of applying a LSTM cell over a RNN cell would be evident, since it is widely known that LSTM cells are capable of keeping information for longer sequences and modeling more complicated relationships in the data.

An interesting and counterintuitive fact is that the attention system used in the encoder-decoder models are not always improving the results. With the little information that is possible to extract from the data, when we use models that are more complex is it not a guarantee that those models will yield better results on the test set. Over-parametrized and complex models are more prone to over-fit, since the model has the capacity to "remember" the training data. Therefore, the model can achieve a small training error but has a bad performance on the test set. Even though the difference between test errors is small, it is possible to see that LSTM and GRU cells have a small performance gain when compared with RNN cells, both on the recurrent and encoder-decoder architectures.

Based on the results we can see that TCN and Wavenet models, both convolutional based methods can match or even perform better than other type of recurrent based algorithms, which is a clear indication that this type of architectures are a viable option to learn sequential dependencies.

5.1.1 Wind Power Generation Hyperparameters

The hyperparameter configuration obtained for the wind power generation forecasting problem is shown in table 5.5. The results are presented in the following order for recurrent models: number of
lags used to train, hidden size, number of layers. For convolutional models the order is: number of lags used to train, number of filters, number of layers and kernel size. In the Wavenet model the results are: number of lags used to train, number of residual connections, number of skip connections, dilation depth and number of repeats.

Table 5.5: Hyperparameter configuration for wind power generation forecasting results.

<table>
<thead>
<tr>
<th>Model</th>
<th>1 hour Parameters</th>
<th>6 hour Parameters</th>
<th>24 hour Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>(1000) – (48) – (1)</td>
<td>(1000) – (48) – (1)</td>
<td>(1000) – (48) – (1)</td>
</tr>
<tr>
<td>GRU</td>
<td>(972) – (59) – (1)</td>
<td>(972) – (59) – (1)</td>
<td>(972) – (59) – (1)</td>
</tr>
<tr>
<td>LSTM</td>
<td>(972) – (59) – (1)</td>
<td>(972) – (59) – (1)</td>
<td>(972) – (59) – (1)</td>
</tr>
<tr>
<td>DRNN - RNN</td>
<td>(992) – (45) – (1)</td>
<td>(992) – (45) – (1)</td>
<td>(992) – (45) – (1)</td>
</tr>
<tr>
<td>DRNN - GRU</td>
<td>(997) – (57) – (1)</td>
<td>(997) – (57) – (1)</td>
<td>(997) – (57) – (1)</td>
</tr>
<tr>
<td>DRNN - LSTM</td>
<td>(994) – (52) – (2)</td>
<td>(994) – (52) – (2)</td>
<td>(994) – (52) – (2)</td>
</tr>
<tr>
<td>TCN</td>
<td>(931) – (10) – (3) – (6)</td>
<td>(931) – (10) – (3) – (6)</td>
<td>(931) – (10) – (3) – (6)</td>
</tr>
<tr>
<td>QRNN</td>
<td>(1000) – (60) – (1) – (2)</td>
<td>(1000) – (60) – (1) – (2)</td>
<td>(1000) – (60) – (1) – (2)</td>
</tr>
<tr>
<td>EncDec RNN</td>
<td>(50) – (10) – (2)</td>
<td>(50) – (10) – (2)</td>
<td>(272) – (57) – (1)</td>
</tr>
<tr>
<td>EncDec GRU</td>
<td>(975) – (58) – (1)</td>
<td>(928) – (60) – (1)</td>
<td>(237) – (60) – (1)</td>
</tr>
<tr>
<td>EncDec LSTM</td>
<td>(42) – (60) – (1)</td>
<td>(42) – (60) – (1)</td>
<td>(999) – (4) – (10)</td>
</tr>
<tr>
<td>Attention RNN</td>
<td>(804) – (10) – (4)</td>
<td>(47) – (14) – (2)</td>
<td>(47) – (14) – (5)</td>
</tr>
<tr>
<td>Attention GRU</td>
<td>(963) – (60) – (2)</td>
<td>(50) – (10) – (2)</td>
<td>(47) – (14) – (4)</td>
</tr>
<tr>
<td>Attention LSTM</td>
<td>(27) – (58) – (1)</td>
<td>(50) – (10) – (2)</td>
<td>(50) – (10) – (2)</td>
</tr>
</tbody>
</table>
5.2 C-MAPSS Turbofan

Now the results obtained with the models introduced in Chapter 3 for the C-MAPSS dataset problem are presented. In this experiment encoder-decoder based models were not used, because the problem do not require to predict more than one step ahead, thus this type of architecture are not relevant for the problem. ADAM and MSE were used as optimization algorithm and loss function, respectively. A 5-fold CV was used, where each split is made based on the number of turbofans. The hyperparameters optimized are shown in Tables 5.6 and 5.7. Unlike the wind power generation forecasting, in this problem the learning rate parameter has an important influence. Thus three different learning rates were tested.

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
<th>Number of lags used</th>
<th>Hidden Size</th>
<th>Number of Layers</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN Architectures</td>
<td>[10, 200]</td>
<td>[10, 256]</td>
<td>[1, 8]</td>
<td></td>
</tr>
<tr>
<td>DRNN Architectures</td>
<td>[10, 200]</td>
<td>[10, 256]</td>
<td>[1, 8]</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.7: Convolutional models search space.

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
<th>Number of lags used</th>
<th>Number of Filters</th>
<th>Number of Layers</th>
<th>Kernel Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCN</td>
<td>[10, 200]</td>
<td>[10, 256]</td>
<td>[1, 8]</td>
<td>[2, 50]</td>
<td></td>
</tr>
<tr>
<td>QRNN</td>
<td>[10, 200]</td>
<td>[10, 256]</td>
<td>[1, 8]</td>
<td>[2, 50]</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: Results obtained for the C-MAPSS dataset.

<table>
<thead>
<tr>
<th>Model</th>
<th>lr = 0.005</th>
<th>lr = 0.0075</th>
<th>lr = 0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>MAE</td>
<td>Score</td>
<td>MSE</td>
</tr>
<tr>
<td>RNN</td>
<td>417.95</td>
<td>15.37</td>
<td>15930.16</td>
</tr>
<tr>
<td>GRU</td>
<td>361.01</td>
<td>13.80</td>
<td>6030.78</td>
</tr>
<tr>
<td>LSTM</td>
<td>175.62</td>
<td>9.06</td>
<td>2613.54</td>
</tr>
<tr>
<td>DRNN - RNN</td>
<td>303.92</td>
<td>13.76</td>
<td>5270.08</td>
</tr>
<tr>
<td>DRNN - GRU</td>
<td>384.83</td>
<td>15.46</td>
<td>7843.49</td>
</tr>
<tr>
<td>DRNN - LSTM</td>
<td>326.01</td>
<td>12.58</td>
<td>5315.37</td>
</tr>
<tr>
<td>TCN</td>
<td>137.53</td>
<td>8.21</td>
<td>1893.87</td>
</tr>
<tr>
<td>QRNN</td>
<td>375.50</td>
<td>13.38</td>
<td>16360.84</td>
</tr>
</tbody>
</table>

Analyzing the results, shown in table 5.8, we can see that the TCN model was the one with the best performance. All models are able to learn the mapping between the sensor data and the RUL, as shown in Figure 5.5. We can also identify that the noisy sensor data is affecting our results, since the predictions are also very noisy.

The theoretical advantage presented by DRNN models is only obtained for RNN cells. This suggests, as previously mentioned, that a vanilla RNN cell is not capable of learning long term dependencies. Using dilated steps is helping the model to learn those dependencies, improving the score in the test
set. GRU and LSTM cells are not improving with dilated steps, which is a hint that the use of dilated steps on this case is not helpful. Dilated steps are making the models unnecessarily more complex, leading to over-fitting problems. This is supported by analyzing the learning curves from the training set shown in Figure 5.6. We can see that using DRNN with a LSTM cell is leading the model to have a much lower training error, which does not happen with RNN cells. This is a clear evidence that the model is over-fitting the training set, in the first case. The sequence length for each engine varies between 50 and 200 and the recurrent architectures are able to learn the long term dependencies solely with LSTM and GRU cells, making the use of dilated steps unnecessary. The advantage of using dilated steps could appear if the sequences were longer.

![Figure 5.5: RUL predictions for different models.](image)

![Figure 5.6: Learning curves for a LSTM and RNN with and without dilated steps. The y-axis are represented in log-scale for ease of interpretation](image)

In this problem we can see that a LSTM cell has yielded better results than the GRU cell, both with a recurrent and dilated recurrent models. Analyzing the performance of the QRNN model we can see that it can match the results from a recurrent model with a GRU cell, even though being a convolutional-
based model. As for the wind power generation forecasting problem we can see that convolutional based methods can outperform recurrent based models. More specifically, the TCN model architecture should be seen as a default option for sequential modeling, because it has the capacity of learning long term dependencies and brings several other advantages for being a convolutional model.

<table>
<thead>
<tr>
<th>Data set</th>
<th>FD001</th>
<th>FD002</th>
<th>FD003</th>
<th>FD004</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSE</td>
<td>153.16</td>
<td>128.83</td>
<td>135.8</td>
<td>141.02</td>
</tr>
<tr>
<td>MAE</td>
<td>8.70</td>
<td>8.08</td>
<td>8.32</td>
<td>8.10</td>
</tr>
<tr>
<td>Score</td>
<td>268.06</td>
<td>634.95</td>
<td>260.56</td>
<td>730.29</td>
</tr>
</tbody>
</table>

In Table 5.9 it is shown the results obtained for the best model, for all datasets. The majority of the bibliography work about this problem have been mostly focused on one of the four data sets. This means that those approaches will be able to learn the patterns that arises solely from that specific data set. Making a model that is capable of generalizing across different operation modes is thus more difficult, but also more relevant. For a benchmark on an unique model across all datasets see [58] and [59]. The results shown in Table 5.9 are the scores obtained for the best model from Table 5.8. The results are slightly better in terms of MAE, in the datasets FD002 and FD004. The fact that the model is biased can be explained by the the higher number of engines in those datasets. Even though the model is biased, it is able to learn the dependencies regardless of the modes of operation and types of failure. Applying a direct mapping between the sensor data and the RUL has the advantage of allowing the user to be agnostic to the problem, and having almost zero knowledge domain about the problem in hand. In Figure 5.7 we can see the results yielded by the TCN model for the four datasets.

![Figure 5.7: TCN model prediction example.](image)

One challenge of these models is the performance dependence with the hyperparameters. Choosing the best hyperparameter set is a hard task, and it is not guaranteed that one can find the best config-
uration. Thus it is important to have a model that has a good performance and is also robust enough
different hyperparameters configurations, and TCN model shows that property. Table 5.8 also shows the
influence of the learning rate in the solution found.

Even though it is common practice, any regularization technique was used in our experiments with
the objective of having the same conditions across all models, to guarantee that we had the same
comparison basis. Techniques such as L1 or L2 regularization and dropout technique [60], can improve
the model capacity to generalize and be less pruned to over-fit to the training set.

5.2.1 C-MAPSS Turbofan Hyperparameters

The hyperparameter configuration obtained for C-MAPSS problem is shown in table 5.10. The results
are presented in the following order for recurrent models: number of lags used to train, hidden size,
number of layers. For convolutional models the order is: number of lags used to train, number of filters,
number of layers and kernel size.

<table>
<thead>
<tr>
<th>Model</th>
<th>Parameters</th>
<th>Parameters</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>(185) – (69) – (3)</td>
<td>(165) – (180) – (4)</td>
<td>(128) – (47) – (4)</td>
</tr>
<tr>
<td>LSTM</td>
<td>(130) – (200) – (3)</td>
<td>(128) – (187) – (3)</td>
<td>(165) – (180) – (4)</td>
</tr>
<tr>
<td>DRNN - RNN</td>
<td>(152) – (56) – (4)</td>
<td>(145) – (67) – (4)</td>
<td>(129) – (100) – (4)</td>
</tr>
<tr>
<td>DRNN - LSTM</td>
<td>(186) – (162) – (5)</td>
<td>(165) – (180) – (4)</td>
<td>(156) – (143) – (3)</td>
</tr>
<tr>
<td>QRNN</td>
<td>(55) – (26) – (3) – (15)</td>
<td>(55) – (26) – (3) – (15)</td>
<td>(117) – (82) – (3) – (4)</td>
</tr>
</tbody>
</table>

53
Chapter 6

Conclusions

Time series analysis were always a field extremely important across several areas, thus it was always under intense development. Supported by the recent advances in the deep learning area the objective of this work was to make a study on deep learning strategies for the time series domain. Initially a study was made about the theory behind the deep learning framework. In this study it was presented which are the standard architectures and what are the default applications for each one of them. The main research question was - Which are the best attributes and architectures to apply in time series prediction problems? With that in mind, a research was made about the architectures that can be applied to sequence modeling and which are the big obstacles and problems with this type of applications. From this, it was assessed what are the key characteristics that a model should have in order to achieve a good performance on time series problems.

Our research concluded that the most promising models in sequence modeling are mostly applied on Natural Language Processing problems. Thereby, some models used were adapted to be applied in time series problems.

To validate and compare the models two different time series problems were used. The datasets were from the wind power generation forecasting and predictive maintenance areas.

The study showed us that recurrent architectures with a LSTM or GRU cells are the default options when we talk about time series, in the deep learning framework. Lately, convolutional based models were being applied to sequential modeling problems with very good results. To asses the viability of applying convolutional based models on time series problems different architectures were chosen to be tested. The performance from both approaches was tested and compared.

The main conclusions are that convolutional-based algorithms, together with recurrent models, should also be seen as a good option to use in time series problems. This is supported by the results obtained on both problems, where we can see that convolutional based models were able to match and even outperform recurrent models.

These are promising results, because a typical problem in sequential modeling, in deep learning, is the challenge of learning long term decencies due to the fact of having a recurrent gradient in the optimization step. Convolutional based methods can avoid this recurrence and be therefore, faster than
recurrent models, and are also more stable in the training phase.

The process of choosing the best hyperparameter combination can be extremely difficult and has a big impact on the results obtained. It was also experienced the problem of over-fitting when we have complex models even though several measures were employed to avoid it, such as early stopping and a CV strategy.

6.1 Future Work

The study performed about deep learning models applied to time series problems showed that RNNs and CNNs are viable options for time series. It was also showed what are the desired characteristics that a model should have to be a viable option. With this in mind the authors propose the following research directions:

- **Further testing** - More datasets with different characteristics should be applied to better understand and further validate the results and characteristics for each model.

- **Regularization techniques** - In this work no regularization techniques were used. This gives a clear margin for improving the results obtained and also to make a study about the impact of those techniques on time series problems.

- **Computational resources** - It is known that one of the big problem with recurrent based models it the time that takes to compute the BPTT algorithm. This is one of the obstacles in sequence modeling. Convolutional neural networks can overcome this problem giving an computational advance over recurrent architectures. A formal study about the gains should be done to evaluate which is the computational gains of convolution over recurrent models.

- **Memory capacity** - A good way to validate the capacity of each model to learn long term dependencies in time series could be doing a formal test on this subject. This is important to try to understand which are the memory capacity limits of each model.

6.2 Contributions

The code developed along this study is publicly available in [61] and [62].
Bibliography


Appendix A

Pipeline

The following figures are a resume of all steps made to get our results. The training strategy is resumed in A.4.

A.1 Preprocessing Stage

![Preprocessing stage diagram](image)

Figure A.1: Preprocessing stage diagram.

A.2 Training Stage

![Training stage diagram](image)

Figure A.2: Training stage diagram.
A.3 Prediction Stage

Figure A.3: Prediction stage diagram.

A.4 Training strategy

Figure A.4: Training strategy flowchart.