TÉCNICO
LISBOA

Intelligent time-series forecasting and event prediction for Predictive Maintenance in IT systems

Pedro Ribeiro Santiago Moreira

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisors: Prof. João Paulo Baptista de Carvalho
Ricardo Carvalho

May 2021

Declaration

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

Abstract

Nowadays, most Information Technology systems already perform Condition-based Maintenance, which provides an overview of a system's condition in real-time and stores its behaviour (in the form of time-series). A Predictive Maintenance approach can use this historical data to apply planning corrective maintenance based on predictions about a system's evolution. This work intends to provide useful research in the scope of Predictive Maintenance through the study of the best approaches and algorithms to perform time-series forecasting and event prediction within the Information Technology domain. The state-of-the-art intelligent methods for time-series modelling were studied, as well as the most promising methods developed for other domains with existent literature ahead of time-series (like Natural language Processing). The main focus was on Machine Learning techniques - from the primary Feed-forward Neural Networks all the way to the more recent and complex Transformers. For the time-series forecasting, none of the experimented models performed satisfactorily. However, it was notable that with the increase of complexity and size of the model's architectures, they learned to output "dummy" forecasts like a naive approach or a constant value, which although not useful, minimise the evaluation metrics. For the event prediction problems, a preprocessing step to detect oscillations in the input datasets significantly boosted the algorithms' performances. Furthermore, the results obtained were not ideal but satisfactory enough to be useful, and the model that showed the best results was the Feed-forward Neural Network. Finally, it is possible to adjust the predictions' sensitivity with the tuning of a data preprocessing factor.

Keywords

Predictive Maintenance; Information Technology systems; Time-series forecasting; Event prediction; Computational Intelligence.

Resumo

Hoje em dia a maioria dos sistemas de tecnologia da informação já executa uma monitorização que permite observar o estado do sistema em tempo real e registar o seu comportamento (na forma de séries temporais). Estes dados históricos podem ser usados para planear uma manutenção preventiva, baseada em previsões da evolução futura do sistema. Neste trabalho foi fetio um estudo em busca das melhores técnicas para modelar séries temporais e prever eventos, no âmbito da manutenção preditiva na tecnologia da informação. Foram analisados os algoritmos inteligentes presentes na literatura para estes problemas, bem como uma adaptação dos algoritmos desenvolvidos noutros domínios, como processamento de linguagem natural, que têm tido mais avanços recentemente. O estudo focou-se em algoritmos de aprendizagem automática – desde as redes neuronais mais simples, como as *Feed-forward*, até aos modelos mais complexos como o *Transformer*. Para a previsão de séries temporais, nenhum dos modelos testados teve resultados satisfatórios, mas pôde ver-se que com o aumento do tamanho e complexidade das redes neuronais utilizadas, estas conseguiam aprender a produzir resultados "cegos", como a *Naive approach*, que apesar de inúteis reduziram os erros das métricas de avaliação usadas. Para a previsão de eventos, uma técnica para detetar oscilações nos dados de entrada melhorou significativamente a performance dos modelos. A rede neuronal *Feed-forward* foi a que apresentou melhores resultados que, apesar de não ideais, foram bons os suficiente para terem utilidade prática. Finalmente, conseguiu-se também ajustar a sensibilidade das previsões de eventos ao regular um parâmetro no pré-processamento dos dados.

Palavras Chave

Manutenção Preditiva; Tecnologia da Informação; Séries Temporais; Previsão de Eventos; Inteligência Computacional.

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
ANN	Artificial Neural Network
AR	Auto-regressive
ARMA	Auto-regressive Moving Average
ARIMA	Auto-regressive Integrated Moving Average
BCE	Binary Cross Entropy
BRNN	Bidirectional Recurrent Neural Network
CI	Computational Intelligence
CBM	Condition-based Maintenance
CPU	Central Processing Unit
FNN	Feed-forward Neural Network
GRU	Gated Recurrent Unit
I	Integration
IT	Information Technology
LSTM	Long Short Term Memory
MA	Moving Average
ML	Machine Learning
MLP	Multilayer Perceptron
MSE	Mean Square Error
MTS	Multivariate Time-series
NLP	Natural language processing
PdM	Predictive Maintenance
RNN	Recurrent Neural Network

RUL	Remaining Useful Lifetime
Tanh	Hyperbolic Tangent
UTS	Univariate Time-series
VARIMA	Vector Auto-regressive Integrated Moving Average

1 Introduction

Contents

1.1 Motivation and Problem description

For any subject of human study, as time goes by, larger amounts of historical data regarding that subject become available – an article from 2018 published in Forbes [10] estimates that the world is producing 2.5 quintillion bytes (2.5×10^{18}) of data every day. The availability of large amounts of historical data makes the forecasting of future events based on the past a possibility and a powerful resource. However, as the data volume grows and the whole environment size and complexity increases, the human brain's capability to understand this data and make predictions upon it is starting to be more and more insufficient. Accordingly, the automation of this process becomes a significant need.

Nowadays, most Information Technology (IT) systems are able to perform Condition-based Maintenance (CBM), which means that they are capable of monitoring the condition of their components in real-time and decide what maintenance is needed. The downside of CBM is that it only orders maintenance actions when certain indicators show signs of decreasing performance or upcoming failure, which means that problems might have already occurred – a problematic system is not reliable, and lack of reliability is not a good sign for enterprise profitability. On the other hand, since IT systems implement CBM, they are already capable of keeping track of their functioning and store their behaviour history. This way, a helpful monitoring approach can make use of this information to predict possible failures in time to avoid/soften them and predict components' future behaviours to, timely, take appropriate precautions. This approach is known as Predictive Maintenance (PdM), which refers to planning corrective maintenance based on predictions about the evolution of a system.

The design of PdM techniques aims to determine the condition of a system and its components ahead of time. By looking forward and knowing what failures are likely to occur, it is possible to schedule adjustments and repairs to apply them before assets fail and/or the system evolves to an unwanted state – these preventive actions will provide a stable environment and an increased assets life. This way, a useful PdM approach shall be capable of providing the means to improve productivity, product quality, and overall effectiveness. Looking from an industry point of view, these improvements achieved by PdM will lead to a vast range of benefits [11] that can both save money and maximize efficiency, such as:

- Reduction (if accurate enough, near elimination) of unscheduled equipment downtime caused by equipment or system failure;
- Better asset management that results in an increased production capacity and labour utilisation;
- Timely routine repairs instead of fewer large-scale repairs;
- Increased equipment lifespan and more economical use of maintenance workers that significantly reduces maintenance costs.

A private company's monitoring software tool for IT systems and components integrated the PdM algorithms developed in this thesis. The private company is - Identity - and this thesis engages in a partnership with them. The monitoring platform performs CBM on metrics such as network utilisation, Central Processing Unit (CPU) load, disk space consumption, etc. As most of IT monitoring platforms, the collected data on these metrics is stored chronologically, with minimal extra computational effort. When a monitoring tool is supposed to be integrated and run by the monitored system, which is the case for most IT systems (and also for the one from Identity), it is imperative to try to keep the computational load as low as possible. Once the goal is solely to provide information about the system in order for it to be appropriately maintained, it should never interact with the systems' actual functioning, or else it will lose its meaning. As such, it is crucial to minimize any impacts of the monitoring platform on the system that runs it. This concern will be addressed later in this dissertation and was taken into account when defining the requirements of a monitoring platform in which the developed PdM techniques can work with. Identity made available for this thesis historical data on hundreds of metrics dating back to more than five years. This data came from a very distinct set of machines and users – be it for internal production inside the company or for external clients/companies that work in different industries and manage systems from very diverse environments. Having this panoply of data available, it is relevant to state that the algorithms developed in this thesis were not tested merely for a single theoretical case and generalised afterwards. Quite contrarily, they were implemented in a functional enterprise software and tested against thousands of data collected from real-world use cases and from a vast range of different scenarios. As requested by the company, this dissertation does not reveal some information such as machine names.

From Wikipedia, [12] "A time series is a series of data points indexed (or listed or graphed) in time order. Most commonly, a time series is a sequence taken at successive equally spaced points in time. Thus it is a sequence of discrete-time data". Since the historical data kept by monitoring platforms from IT systems is often stored chronologically and with a timestamp attached to every collected metric, it seems evident that this data can be looked at and analysed as time-series. Time-series analysis is one of the areas with the biggest potential in PdM, and historical data, which is its feedstock, can be acquired very easily by IT systems – most of them already perform this data collection with the implementation of CBM. The use of Artificial Intelligence (AI) and other intelligent methods in PdM has been growing in the past few years, but the majority of the companies did not adopt it yet [13]. This way, the research on intelligent methods to analyse time-series in a PdM perspective can be not only a very recent and interesting development, but it can also have a significant practical impact in the real-world environments.

There are several different approaches that one can take towards the development of a PdM solution. After a PdM background study applied to modern IT systems needs, and a strategic fit with the Identity's monitoring software, this dissertation will tackle two separate problems in the PdM world that were found to be the most convenient and advantageous to be integrated with the monitoring of IT systems:

Time-series forecasting – this solution will consist in the development of methods to predict the future values of an IT system's metric, based on its past values and the past values of multiple other metrics (the system's history), for example, predicting the CPU load for tomorrow at 2 pm.

Event prediction – this solution will consist in the development of methods to predict the occurrence of sporadic events before they actually occur, based on the event's previous occurrences, some others events previous occurrences and the history of some system's metrics, for example, issuing a warning four hours before the system collapses due to lack of free memory.

After having these two functionalities implemented and working at trustworthy levels, a system could be adequately maintained, operating at its maximum potential, durability, and providing a reliable service given that one can now avoid unexpected problems ahead of time.

1.2 Topics Overview

1.2.1 Time-series Forecasting

A time-series is a sequential set of data points, usually measured over successive times and recorded chronologically. Mathematically, it can be defined as a matrix $\boldsymbol{X}$, where each of its elements $\boldsymbol{x}^t \in \mathbb{R}^m$ and where $\boldsymbol{x}^t$, in practice, is an m-dimensional array that represents a set of m arbitrary points collected over time t, $\boldsymbol{x}^t = [x_1^t, x_2^t, ..., x_n^t]^T$. Thus, matrix $\boldsymbol{X}$ can be described by

$$\boldsymbol{X} = \left[\boldsymbol{x}^1, \boldsymbol{x}^2, ..., \boldsymbol{x}^n\right]^T . \quad (1.1)$$

For a formal definition of a time-series, one can consult [14].

Time-series analysis stands for the methods for analysing a time-series dataset so that, based on the available data (the observations from the past), one can create a model, by extracting meaningful statistics and other important characteristics, that correctly shape the series. Time-series analysis is not a novelty. It is a topic with many research and many underlying areas, one of which is time-series forecasting, which is the one with the biggest relevance and applicability for this study. Time-series forecasting consists in the methods that allow the prediction of (unknown) future observations of a time-series. An example of time-series forecasting is in Figure 1.1, where the red line represents the forecast performed to the (pre-existing) time-series represented by the blue line.

One important discrimination while introducing time-series is the difference between Univariate Time-series (UTS) and Multivariate Time-series (MTS).

UTS, as the name implies, is a time-series dataset with a single time-dependent variable. Therefore, for each period, there will be a one-dimensional value. An example of a UTS dataset of a CPU load percentage along time, with a period of one minute per sample, can be seen in Table 1.1.

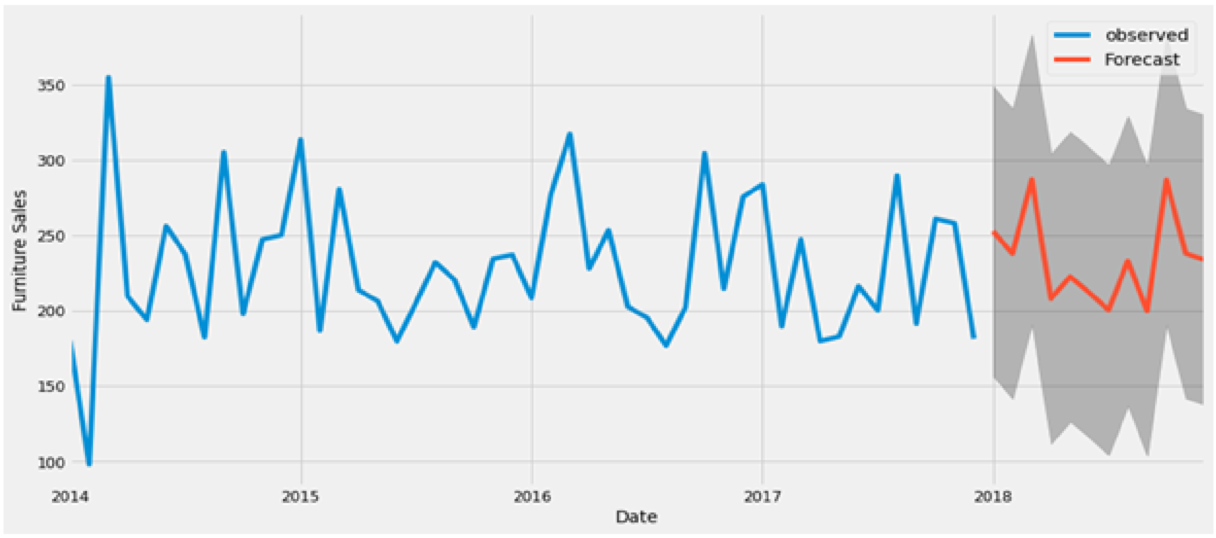


Figure 1.1: Example of a time-series forecasting (taken from [1]).

Table 1.1: UTS dataset example of CPU load [%].

Time	CPU load [%]
15h 03min 00s	15.32
15h 04min 00s	18.01
15h 05min 00s	15.44
15h 06min 00s	23.26
15h 07min 00s	30.98
15h 08min 00s	20.05

MTS, as the name also implies, is a time-series dataset with multiple time-dependent variables, N. Therefore, for each period, there will be N values, or a N-dimensional sample. An example of an MTS dataset, with $N = 3$, and a period of one minute, can be seen in Table 1.2. Notably, Table 1.2 is an extension of Table 1.1 where, beyond the CPU load percentage, it is also collected simultaneously: free memory percentage, and the number of active processes.

Table 1.2: MTS dataset example.

Time	CPU load [%]	Free memory [%]	Number of processes
15h 03min 00s	15.32	63.35	163
15h 04min 00s	18.01	71.27	205
15h 05min 00s	15.44	54.20	209
15h 06min 00s	23.26	63.99	234
15h 07min 00s	30.98	65.83	210
15h 08min 00s	20.05	66.24	266

When asked to forecast the CPU load percentage for the next (unknown) periods, an MTS analysis approach will not only take into account the past values of the forecasting variable but also the other metrics that were collected alongside. Such a method would try to learn the dependency of the other metrics on the CPU load percentage and forecast it accordingly.

It is also worth mentioning that while the methods studied and developed in this dissertation focused on MTS, research on the state-of-the-art UTS approaches also took part. However, the experiments comprising these methods (after contracting the datasets to UTS) never led to superior results. Hence, the results do not include them.

1.2.2 Event Prediction

An event is a real-world incident that takes place at a particular time and can be objectively discriminated (somehow labelled or described). Different types of events can range from large-scale (like a natural disaster) to small-scale ones (like blowing up a tire), as well as vary from more conceptual (like a software bug) to more practical occurrences (like an electrical short circuit). Defining an event mathematically can be messy and is hardly intuitive, but for a scientific definition of an event one can consult [15].

Event analytics is a broad area with applications in a plenitude of different domains. One of these applications is event prediction, which is the one that this study will focus on. Event prediction targets the identification of future events before they actually happen, based on historical information from the past. Event prediction is not a concept as well established as time-series forecasting, so its understanding might differ slightly from different sources: (a) some define it as a matter of if/when – foreseeing when a particular event is going to take place (or if it will never take place at all); (b) some define it as a matter of what – what event(s) will occur at a specific time in the future; (c) some define it as a matter of where – predicting the location of a certain event (or set of events) that will occur at a specific time in the future; (d) and some may define it as other variants or as the union of the several ones. In this dissertation, the first definition will be the one used when referring to event prediction.

The event prediction problems tackled in this dissertation are referent to problems triggered in the monitored IT systems. Every distinct problem is identifiable and labelled to an event – when an event occurs, it means that a problem was triggered. Similarly to the time series datasets, every event occurrence will come with a timestamp attached stating when the event occurred. However, instead of having periodically collected samples and thus, one value for every period, there are only values when an event occurs, which is supposedly random and without any respect for periods. In order to have these datasets interpreted and analysed as time-series, they need some data preprocessing – those procedures will be explained later in Chapter 4. Events datasets will be in the form of Table 1.3 where, when a problem is triggered (event occurrence), a the platform stores a value of 1 with its correspondent timestamp; when the problem is solved (or stopped existing), it stores a value of 0 with its correspondent timestamp as well.

In event analytics is often hard to have a reliable representation of events characteristics represented by plots; on the other hand, tables usually provide a better visual understanding of events behaviours and occurrences.

Table 1.3: Event dataset example.

Timestamp	Event
2020-01-05 01:00:03	1
2020-01-05 01:07:29	0
2020-01-08 04:10:12	1
2020-01-08 08:05:17	0
2020-01-09 22:43:08	1
2020-01-10 01:13:05	0

1.3 Claim of contributions

One can look at the work developed in this dissertation as two separate sections – time-series forecasting and event prediction – and independently analyse its contributions and results. This way, for each of the sections, this thesis has the following contributions:

Time-series forecasting – the study of the most promising methods present in the literature for this type of problem; even though this work did not find any approach with satisfactory results in this section, it was possible to gather some learning characteristics of each of the intelligent models and understand what they try to learn from the input data.

Event prediction – the study of the most promising methods and adaptation techniques present in the literature for this type of problem; despite the non-ideal performances, the event prediction experiments showed results that can already have a practical use. A data preprocessing technique significantly improved the performances, and one of the studied intelligent algorithms stood out among the others with better results consistently. Moreover, a data-tuning technique is capable of projecting a deliberate bias in the predictions of the events.

1.4 Thesis Outline

After introducing the relevance of PdM in IT systems and its applicability in the industry, this chapter - Chapter 1 - provides a presentation of the two topics tackled in this dissertation, namely: (a) time-series forecasting; (b) event prediction using time-series .

Chapter 2 reports the technologies and related studies reviewed in order to investigate the best suited state-of-the-art methods for the covered topics.

Chapter 3 provides a theoretical background of the state-of-the-art intelligent methods that promise, according to Chapter 2, to be the most promising in solving the two tackled problems.

Having explained the architectures of the intelligent methods experiment, Chapter 4 describes in detail their implementations and the experiments conducted in this work.

Chapter 5 illustrates the results obtained with the experiments explored in this dissertation and discusses the possible interpretations that one can take from them.

Finally, Chapter 6 aims to wrap-up this dissertation by summarizing the problems tackled and a broader view of its conclusions. It suggests the future work that one can embrace to continue the work developed in this dissertation and objectively improve it.

2

State of the art

Contents

2.1 Time-series Forecasting

Time-series analysis and forecasting is not a new concept, it is a broad topic with a long history of research and investigation. However, for a long time, the classical forecasting methods (classical is the term commonly used to refer to the statistical and linear approaches) took over this domain. According to [16,17], up until the late 1970s, classical approaches such as Auto-regressive Integrated Moving Average (ARIMA) were the undisputed state-of-the-art technologies in time-series forecasting. As time went by, it became clear that these classical models were not thoroughly capable of adapting to many real-world situations. In order to solve this incapacity and to develop better forecasting models, approaches based on Computational Intelligence (CI) – covering methods like Artificial Neural Network (ANN), Fuzzy Systems and Evolutionary Computation – are being more studied in recent decades [18–20] and have proven to be more effective in several domains, as concluded in [21]. However, an acknowledged conclusion in all the literature reviewed for this dissertation, regarding time-series forecasting, is that there is no approach proven to be better than all others for every scenario or use case. Thus, the classical methods are always worth being tested and evaluated. As illustrated in a recent overview across the existing statistical forecasting methods for time-series forecasting [22], the method that revealed to be the most accurate across more domains and datasets (by a large margin) was ARIMA or slight extensions of it. This way, and as this thesis dug deeply over Machine Learning (ML) models, making them its main research focus, the only statistical model tested in this work was ARIMA. Further research on this work could include other state-of-the-art statistical methods to be compared with ARIMA (and the other non-statistical methods), or possibly replace as a module of a broader architecture.

2.1.1 Statistical models

An advantage usually stated in favour of some statistical models, such as ARIMA, is that beyond having a high accuracy in many study cases, it is relatively robust and straightforward when compared to more complex CI methods like ANN. Consequently, it might be easier to understand its results and tune the models accordingly. Furthermore, many users without the required expertise to develop and adapt complex CI models will also opt for ARIMA.

One clear benefit of the classical (and simpler) linear methods is that they are able to perform well when the data volumes are slim [23] since when comparing to most CI methods the number of parameters to estimate is utterly low. On the other hand, with a low number of parameters to learn from a time-series, it may not be able to model complex datasets and perform complex forecasts. Accordingly, a significant liability pointed to ARIMA in [17, 22] is that while it can show good performances for UTS datasets, it does not scale well to MTS and its accuracy can suffer significantly. To overcome this difficulty, vector-generalised approaches of the statistical methods, like Vector Auto-regressive Integrated

Moving Average (VARIMA) were proposed, and revealed better results [24]. However, none of these vector-generalised techniques is suited to capture nonlinearities in the datasets and will probably not succeed when modelling more complex behaviours. This is due to the fact that these techniques only take into account the inherent features of one series (each time-dependent variable of the dataset) at a time, and thus are not well suited to capture cross-series relations. Beyond that, since these methods build one model for each series, frequent retraining is often required and can become computationally too expensive for large databases. For these reasons, significant effort has been (and is still being) put into non-linear models like regime-switching [25]. The remaining problem of these alternatives is that they require the application of predetermined nonlinearities and tend to fail for different MTS datasets (where the nonlinearities will most likely differ). Since for PdM purposes (and for the datasets used in this work), the algorithms require learning capabilities from several different metrics, this limitation is a major drawback.

Another limitation pointed to ARIMA, is their inability of maintaining good performances when forecasting several time-steps ahead (multistep-ahead forecasting) [26, 27]. CI methods tend to be the unanimous choice for this type of problems.

2.1.2 Computational Intelligence and Machine Learning models

ML models have been proposed in academic literature as heavier alternatives to the statistical ones, with the aim of being more accurate and with the ability to learn more complex patterns. However, the research over unbiased comparisons between ML and statistical methods is very scant. In [28, 29], such comparisons can be seen but, despite being very recent (2018 and 202, respectively), in both is clearly stated that the experimental data on this topic is yet scarce and needs more investigation to be conclusive.

Regarding the comparisons of the different ML algorithms themselves, again the academic literature is often not conclusive. The objective comparisons available between ML methods tend to be either too old, like in [30, 31] (and plenty of new ML methods have come out and become popular since then), or too limited to a specific domain, as in [32, 33] which will lead to misleading outcome expectations when tested in different environments and different datasets. Furthermore, rarely these specific domains are IT related, which is precisely the focus of this dissertation.

Another limitation found in this work's academic literature investigation was the lack of research on forecasting IT systems metrics. While some other fields already have an extensive study of a vast range of ANN models and some benchmarks and base results defined, for example, the energy sector [34–36] and financial instruments like stock prices [37, 38]. Contrarily, for IT systems in particular, there is still a long way to go. As it can be seen in the few existing papers on this topic like [39], the "Literature Review" section bases its findings on work conducted in other fields and environments, to afterwards generalise it for IT systems. For the same reasons, most of the related work reviewed in for dissertation covers datasets outside of the IT domain.

2.1.2.A Artificial Neural Networks (ANN)

As the size and complexity of the available data increased in recent years, the ANN have become more popular and, arguably, the most dominant methods for prediction tasks [16]. The most basic type of ANN is the Feed-forward Neural Network (FNN). Contrarily to other ANN models, like the Recurrent Neural Network (RNN) which integrate feedback loops, the FNN only comprises forward connections among its neurons, being this way the simpler and computationally lighter type of ANN. Given its popularity for decades, there is a vast abundance of research available studying the behaviour and characteristics of FNN in forecasting applications. Such an example can be seen in [40]. The authors only refer to ANN in their work, however, at the time, the existent ANN architectures were far less, so it was reasonable to generalise the conclusions from FNN experiments. Nevertheless, all the ANN architectures studied are somehow based on the ANN and thus, maintain their properties. Back to [40], the authors acknowledge that ANN hold many learning attributes that make them the obvious candidates to replace and compete with the classical techniques for time-series forecasting – the ANN are designed to model any pattern from a dataset without the need of predetermined parameters; the ANN are universal approximators, which implies that they are capable of learning every pattern and relation possibly present in a dataset, particularly the non-linear relationships [41] which are the hardest drawback of the classical methods.

In [42], the authors place ANN against ARIMA over forecasting tasks and conclude that the first becomes significantly superior as the forecasting horizon increases. The authors also state that the benefits of the ANN become clear when learning from time-series of different complexities – “We have found that for time series of different complexities there are optimal neural network topologies and parameters that enable them to learn more efficiently” – this conclusion inherently brings the downside of having to find the best topology and parameters (the ANN architecture and the numbers of layers and neurons) for a learning task. To obtain maximal accuracies from ANN, it is also important to tune the best training procedure. Another critical decision is the choice of the input variables (features) to feed the models, as highlighted in [43] where it is exemplified how the right choice of input-variables can affect the model’s performance.

In [44], the authors proposed an ANN model for electric load forecasting, combined with the “similar days approach” – this technique is based on searching historical data of days of one, two or three years having the similar characteristics to the day of forecast. The characteristics include similar weather conditions, similar day of the week or date. This data arrangement presented very promising results in the cited paper. However, this method requires quite some manual intervention and would hardly be integrated with a product and perform in a fully automated way. The methods studied in this dissertation were designed and integrated into a monitoring platform that required them to be fully automated since they are intended to act on a vast range of different IT metrics.

The introduction of ANN models in the forecasting literature has not overtaken the classical methods because they both have their pros and cons, as already stated. Accordingly, hybrid approaches have emerged. In [45], the authors proposed a hybrid model for forecasting, which combines an ANN with ARIMA with the intent of leveraging both of their virtues. While he ARIMA model was used to capture the linear components of the time-series, the ANN modelled the so-called “residual” components of the series, which stands for the nonlinearities. This approach has been found to be very efficient in reducing biases in forecasts.

To feed ANN with time-series datasets, the most common method is to split the data in chunks of sequential and consecutive windows with the length of the ANN input layer. The ANN is then usually trained to predict the following window (multistep-ahead forecasting) or the single data point immediately after the input window (onestep-ahead forecasting). The limitation of the basic ANN, the FNN, is that by only having forward connections, every new input is considered in isolation. So the network cannot take into account the temporal order of the inputs. The RNN architecture came to solve this problem.

2.1.2.B Recurrent Neural Networks (RNNs)

By integrating feedback loops in the network design, the RNN is specialized to take into account the order and temporal patterns of data. For this reason, the RNN is the most widely used ANN for sequence prediction tasks. In [46], it is well proved and explained how the RNN succeeds in capturing the order patterns of a sequence. Every RNN comprises multiple cells, usually, each one accountable for each input of one input sequence. By far, the most popular cells are the following three: (a) Elman RNN cell, introduced by J. L. Elman [47] and usually referred to as the default RNN cell; (b) the Long Short Term Memory (LSTM) cell, introduced by S. Hochreiter and J. Schmidhuber [48]; (c) and the Gated Recurrent Unit (GRU) cell, introduced by K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio [49]. The theoretical differences and are described in chapter 3. Other cells have been proposed and tested over time, like in [50] but have not shown functional improvements. Nonetheless, rarely these proposed cells have been tested with time-series in particular. Regarding comparisons of the three popular cell types, the literature is extensive and complete. In [51], those three models and two others (Echo State Networks and the Nonlinear Auto-regressive Network with Exogenous inputs) are compared in a short-term load forecasting problem. The authors concluded that both LSTM and GRU showed similar performances across all the tests. There is no hard evidence to believe one is going to outperform the other without prior testing. The Elamn RNN cell is considerably faster to train and has a similar performance in many datasets. On the other hand, for datasets where the temporal dependencies are highly nonlinear and abrupt, it shows a notable drop in performance when compared with other two. Moreover, the authors indicated that all the RNN models (all the three cell types) show significantly longer training times when comparing to other architectures. This is due to the backpropagation through time (needed to train the feedback loops) which is very time-consuming and computationally expensive.

Several RNN based architectures have been developed in recent years, and even though almost all were developed for Natural language processing (NLP) problems, many have been experimented with time-series datasets and revealed promising results. The state-of-the-art study conducted in this work found that the most frequent RNN architecture for time-series forecasting problems is the staked architecture, which basically stacks up multiple layers of RNN on top of each other. A recent work [23] used this architecture for time-series forecasting problems and revealed good results in the CIF 2016 and NN5 forecasting competitions. Due to the vanishing gradient disadvantage that occurs with the Elamn RNN cells (explained in chapter 3), the authors used LSTM cells.

Since [52], several Bidirectional Recurrent Neural Network (BRNN) based approaches have been proposed and tested over time and with promising results in domains like NLP. These techniques connect two hidden layers of opposite directions and have the advantage of having every cell processing its respective input, taking into account not only information from the previous inputs but also from the future ones. The literature of BRNN models applied to time-series is minimal, but mainly because the additional benefit of these architectures over the classical RNN (which is taking into account the future inputs) does have the same impact on time-dependent variables. For NLP tasks, it is intuitive to state that, by looking at a sentence, the last words can affect the meaning of the previous ones. This way, having a model that processes each word of a sentence taking into account all the rest can bring considerable advantages to the model performance. On the other hand, in a time-series data set, such mechanisms do not apply anymore, and thus there are not many reasons to believe that BRNN would improve the performance of the models over simple RNN.

2.1.2.C Sequence-to-sequence models

Recently introduced by [53], the sequence to sequence models have also been growing a lot in popularity and are, arguably, the latest state-of-the-art technology for sequence prediction tasks. This contemporary ML technique classically comprises an encoder and a decoder, both acting as independent RNN architectures. For a detailed explanation of these methods refer to chapter 3. It is relevant to state that, just as most part of the newly developed ML methods, the sequence-to-sequence models were designed for NLP purposes, namely, applications like language translation, image captioning, conversational models and text summarization. As such, the vast majority of the literature is applied to these purposes and the available studies comprising these models and time-series datasets is scarce and not broad enough to be conclusive.

In [54], a sequence-to-sequence model using LSTM cells in both the encoder and decoder was tested for a time-series dataset of a cloud server workload – precisely the type of problems targeted in this dissertation. This novel approach was compared with vanilla RNN models (with the three popular cell types) and with the state-of-the-art statistical methods and outperformed them in the two datasets tested (the Google clusters dataset and the Dinda dataset from Unix systems).

The most popular add-on to the sequence-to-sequence models is the attention mechanism. Two different examples of attention mechanisms were proposed in [55, 56]. This approach was designed to overcome a liability of the vanilla sequence-to-sequence models – the encoder encodes the whole input sequence into one vector, which is later decoded by the decoder. In [57], the authors prove how this encoding leads to information and context losses. This way, the attention mechanism aims to identify the encoded vector's context and set the decoder to "pay attention" to the relevant parts of it. Furthermore, the attention mechanism uses a softmax distribution in the decoder to assign how much relevance should be given to each part of the encoder's outputs. However, for long sequences, the softmax outputs may not be sparse enough. In [58] another activation function was proposed – sparsemax – to solve this problem, and revealed significant improvements for multi-label classification problems. The attention mechanism, as well as the discussed distributions, are covered in chapter 3.

In the three cited papers [55–57], and several others, this add-on has been empirically proven to improve the results in NLP tasks. The same can not be stated for time-series forecasting problems, as the experiments on this field are scarce. Nonetheless, there is no reason to believe that the improvement given by the textual context in NLP tasks cannot translate to time-series components like seasonality and thus, such alternatives should be tested.

Since the traditional sequence-to-sequence models with the attention layer require training RNNs plus the attention weights for the whole sequence at each time step, it ends up being computationally very expensive. With the intent to solve this, "a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely" was proposed. The famous work "Attention is all you need" [9], proposes this (self-attention based) Transformer architecture for sequence modelling and has achieved great successes since it was launched. The Transformer's architecture is covered in chapter 3. Many different works already came out applying this model to several tasks like music modelling [59], image generation [60], speech recognition [61], etc. Analogously to the other sequence-to-sequence models, the Transformer has yet not been covered properly for time-series problems, but it has significant potential to succeed in it. However, it must be noted that the space complexity of the self-attention mechanism grows quadratically with the sequence length of the inputs. This complexity can become computationally challenging if the time-series dataset holds relevant long-term dependencies, because it would mean that the input sequence would as well have to be long enough to comprise all the information required to catch those dependencies.

2.2 Event Prediction

Event prediction problems take part in a vast range of classical problems such as anomaly detection, Remaining Useful Lifetime (RUL) prediction and many others that have become popular, and over which the available literature is considerable. However, it must be taken into account that these classical problems might comprise broader perspectives of which event prediction is only a module, or need adjustments to become applicable to the event prediction engaged in this dissertation. For example, the anomaly detection study conducted in [62] proposes a model to predict an earthquake occurrence. This process could be divided into two segments: (a) determine the geographical location of the future earthquake (b) and predict if/when it was going to be triggered. The relevant process to retain for this work would only be the second. Another example can be given after [63], where the authors proposed a RNN with LSTM cells to predict how much longer lithium-ion batteries would last. Here the event is well defined, but a binary prediction of yes or no is not clear, as the predicted value will be a number (of days). This problem could be adapted though; if instead of "how much time", the computed prediction meant to say, at every time-step, if the battery would fail. If such a change of perspective is possible, [63] is now aligned with this dissertation's event prediction problems. One last example is the anomaly detection study conducted in [64], where, instead of forecasting a specific error or event, the aim is to detect anomalies, as any out of the ordinary behaviours, in continuous variables. Consequently, [64] is a different type of study, with no visible applications to this dissertation's goals.

Contrarily to forecasting problems, in the event prediction domain (for time-series datasets), the linear statistical methods have been practically outdated since the appearance of ML methods, especially for binary event predictions. Given that the outcome (one or zero) can not be seen as a continuation or combination of the input variables, it makes sense that the linear approaches might not be very successful.

Given that the intelligent methods that aim to perform event predictions will also attempt to model the input dataset to take meaningful conclusions out of it, it makes sense that most of the review covered for time-series forecasting can also be, to some extent, extrapolated for event prediction approaches that learn from time-series (which is the approach tackled in the dissertation). For example, stating that FNN does not take into account the temporal order of the inputs or that the LSTM cells overcome the vanishing gradient problem seen in the vanilla Elman RNN cells, are conclusions that also applicable if the desired output is the prediction of an event and not the forecasting of one of the input variables.

Furthermore, this extrapolation of domain would be no novelty. In [65], the authors performed a comparison of the most popular time-series forecasting methods in the tasks of event prediction and RUL. Evidently, some learning procedures adaptations we necessary like changing the loss functions and tuning the activation functions of the ML models, as well as changing evaluation metrics (for example, instead of measuring accuracy, measure the precision and recall).

In the recent work [66], is conducted a systematic survey over event prediction existent techniques over its several domains and datasets, including time-series. It can be seen that for time-series problems in particular, the evolution of the state-of-the-art ML algorithms is very similar to the one in the study conducted for time-series forecasting in Section 2.1 – starting from FNN, all the way to encoder-decoder RNN architectures with attention mechanisms. The authors also emphasized the importance of the right approach towards the input data, in alignment with the type of problem and type of dataset. One of the examples covered, applicable to this dissertation, is the prediction of "rare events" in the anomaly detection domain. Rare events problems often lead to highly imbalanced datasets, which require either a specialized data approach to normalize the dataset or some tuning on the network's learning methods. For these problems, the authors [66] concluded that the anomaly detection techniques such as one-classification, like the model proposed in [67] and hypothesis testing [68, 69], were the most common. However, for time-series datasets that will feed ML model, the most common approach to deal with highly imbalanced datasets (and the one that has proven to lead to better results) is by artificially oversampling the minority class occurrences of the dataset (or undersampling the majority class occurrences) [70].

3 Theoretical Background

Contents

3.1 ARIMA

The ARIMA is a linear process; as such, it models the past observations of a given time-series variable with a linear function and forecasts its future values using that function. This way, the variables supplied to the ARIMA algorithm are presumed to be linear. The ARIMA model is based on the Box-Jenkins methodology and it comprises both an Auto-regressive (AR) and a Moving Average (MA) models.

The AR model defines the present/future time-series observations as a linear combination of its past observations, taking into account a noise term. This way, the AR function of order p, $AR(p)$, can be mathematically described by

$$Y_t = \sum_{j=1}^{p} \beta_j Y_{t-j} + \varepsilon_t, \tag{3.1}$$

where p is a non-negative integer, $\beta = \{\beta_1, \beta_2, ..., \beta_p\}$ is the vector holding the AR model coefficients and ε_t is the forecast error (the noise term). Y_t is predicted value for time t, that only depends on its own lags $\{Y_{t-1}, Y_{t-1}, ..., Y_{t-p}\}$.

The MA model works by analysing how wrong the AR predictions performed in predicting previous time-periods and estimates an adjustment to work as a correction for the current time-periods. Essentially, the MA model uses lagged values of the forecasting error to improve the current forecast. This way, the MA function of order q, $MA(q)$, can be mathematically described by

$$Y_t = \varepsilon_t - \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}, \tag{3.2}$$

where ε follows a normally distributed sequence of random white noise with a null mean and a constant variance: $\varepsilon \sim N(0, \sigma^2)$ and θ is the vector holding the MA model coefficients: $\theta = \{\theta_1, \theta_2, ..., \theta_p\}$. Now, Y_t only depends on its own lagged forecast errors $\{\varepsilon_{t-1}, \varepsilon_{t-1}, ..., \varepsilon_{t-q}\}$.

The AR and MA models can be combined together to form the Auto-regressive Moving Average (ARMA) model. This way, the ARMA function of order p and q, $ARMA(p, q)$, can be described by

$$Y_t = \sum_{j=1}^{p} \beta_j Y_{t-j} + \varepsilon_t - \sum_{j=1}^{q} \theta_j \varepsilon_{t-j}. \tag{3.3}$$

The limitation of the ARMA process is that it assumes that the time-series to model is stationary, i.e. the mean and the variance are constant over time (seasonality does not exist). As such, stationarity is a must to take advantage of this model. For non-stationary time-series a transformation procedure is required to eliminate the non-stationary components before applying the ARMA models. The most common transformation is a differencing procedure that computes the difference between consecutive time-steps, as mathematically described by

$$Y_t^{'} = Y_{t+1} - Y_t. \tag{3.4}$$

Differencing stabilizes the mean of the time-series by removing the changes in the level of the series, eliminating the trend and seasonality components. But one differencing might not be sufficient, it can necessary to repeat the procedure d times, where d will be the order of differencing:

$$\begin{aligned} Y_t^{'} &= Y_{t+1} - Y_t, d = 1 \\ Y_t^{''} &= Y_{t+1}^{'} - Y_t^{'}, d = 2 \\ Y_t^{'''} &= Y_{t+1}^{''} - Y_t^{''}, d = 3 \\ &\ldots \end{aligned} \tag{3.5}$$

In the ARIMA model, d represents the Integration (I) order. Therefore, the ARIMA model can be written as $ARIMA(p, d, q)$, where p represents the AR process order, d represents the order of the I process (or the stationarity order), and q represents the order of the MA process.

3.2 Artificial Neural Network

3.2.1 Feed-forward Neural Network

Also knows as Multilayer Perceptron (MLP), the FNN was the first ANN architecture to come out and, as the name implies (feed-forward), the information flows all the way through the network with only one direction, starting from the input nodes, passing through the hidden nodes (if any) and finishing at the output nodes. There are no cycles or feedback loops.

The so-called nodes that compose the network – artificial neurons (or single layer perceptrons) – receive a set of inputs that is passed through a respective set of weights (which are the trainable parameters of the network). The resultant set is summed up and passed through an activation function. The mathematical function of an artificial neuron is defined by

$$a = \sigma \left(\sum_{i=1}^{n} w_i \cdot x_i \right), \tag{3.6}$$

where $\boldsymbol{x} = \{x_1, x_2, ..., x_n\}$ is the input vector, $\boldsymbol{w} = \{w_1, w_2, ..., w_3\}$ is the trainable vector of weights, σ is the activation function and a is the resultant output of the perceptron. A schematic representation of the typical artificial neuron is represented in Figure 3.1.

It must be noted that a bias term is usually added to the network, which consists of a unitary value appended to the inputs and with respective a fixed weight representing the bias value. In Equation (3.6),

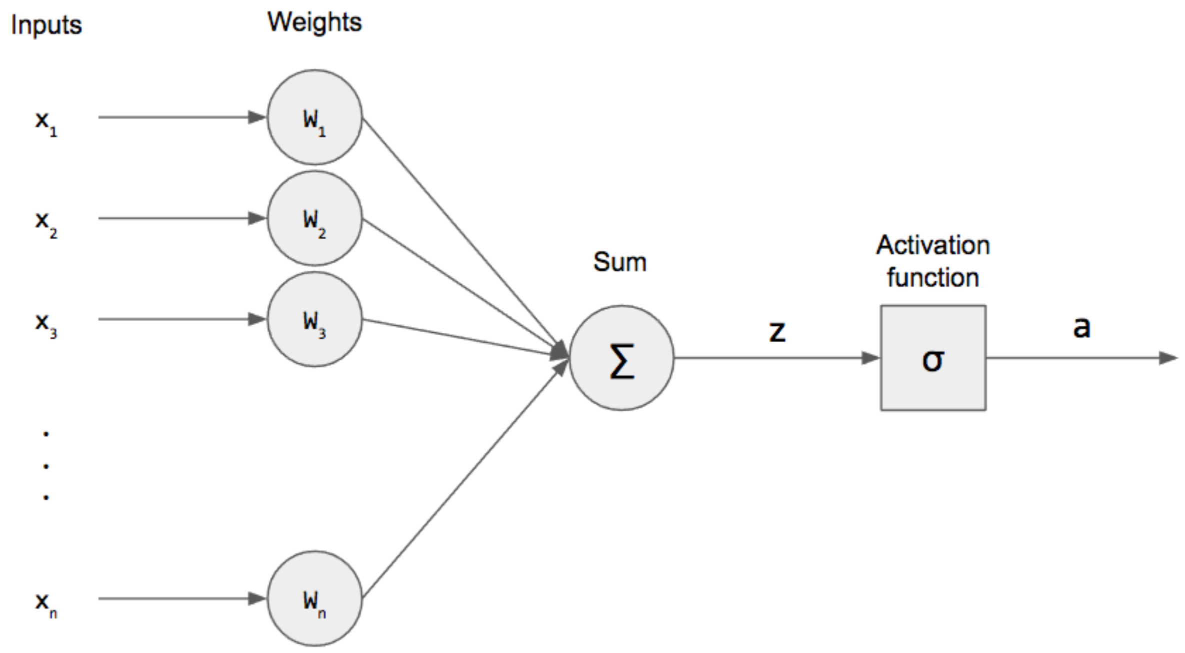


Figure 3.1: Single layer perceptron (taken from [2]).

this bias term would consist in an extra term for $i = 0$ with the input $x_0 = 1$ and the respective weight $w_0 = bias$.

Now, a FNN architecture comprises several of these artificial neurons organized in layers and, typically, every neuron is connected to all the neurons of its adjacent direct layers. A three-layered FNN example can be mathematically described by

$$y = a^{(3)}\left(a^{(2)}\left(a^{(1)}\right)\right), \tag{3.7}$$

where y is the output of the network and $a^{(l)}$ represents the outputs of the activation functions of the artificial neurons that compose layer l. A schematic representation of the such a model is represented in Figure 3.2, which can also be generalized to represent the classical FNN architecture.

In practical terms, the goal of a ANN is to find the set of weights $\boldsymbol{w}$ for the function of Equation (3.7) that will have it replicating as best as possible a particular real-world function. This is achieved by training the weights based on a set of input-output sample pairs – a training dataset that should tell the ANN what should be outputted given a certain input. This type of training is defined as Supervised Learning and it applies to all the ANN architectures presented in this work.

The activation functions of the artificial neurons that compose a ANN introduce nonlinearities in the model. These elements are necessary to have the network capable of approximating every possible function, specifically the nonlinear ones. The activation functions used in this work were four of the most common in this domain:

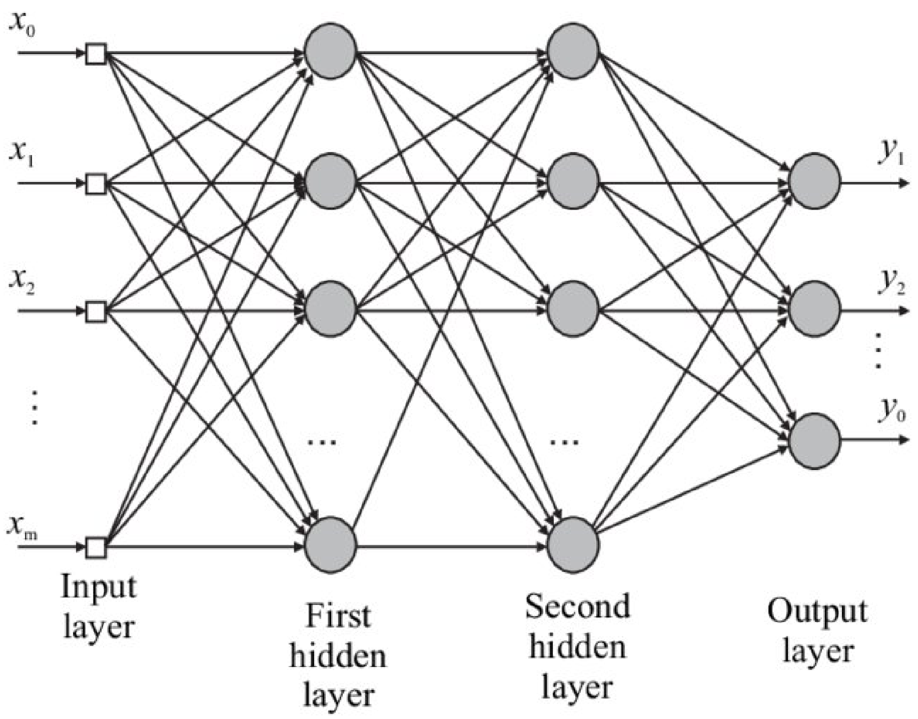


Figure 3.2: Feed-forward Neural Network example (taken from [3]).

- Linear unit – mathematically described by

$$f(x) = x; \tag{3.8}$$

- Rectified Linear unit (ReLu) – mathematically described by

$$f(x) = max(0, x); \tag{3.9}$$

- Sigmoid unit – mathematically described by

$$f(x) = \frac{1}{1 + e^{-x}}; \tag{3.10}$$

- Hyperbolic Tangent – mathematically described by

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{3.11}$$

3.2.2 Recurrent Neural Network

One limitation pointed to the FNN architecture is its intrinsic incapability of processing sequential data. Given that the aforementioned network "treats" equally all of its inputs, it does not take proper advantage of their order.

Specifically designed for sequential data, the RNN is a type of ANN that puts the connections between its nodes to form a directed graph along a temporal sequence, which results in a temporal dynamic behaviour. Derived from the FNN, the RNN is a type of network that incorporates an internal state (that acts as a memory), which is the component that empowers its capability of processing sequences of inputs. A graphical representation of a general RNN architecture is present in Figure 3.3. The internal state of a cell, denominated hidden state, is represented by h_t.

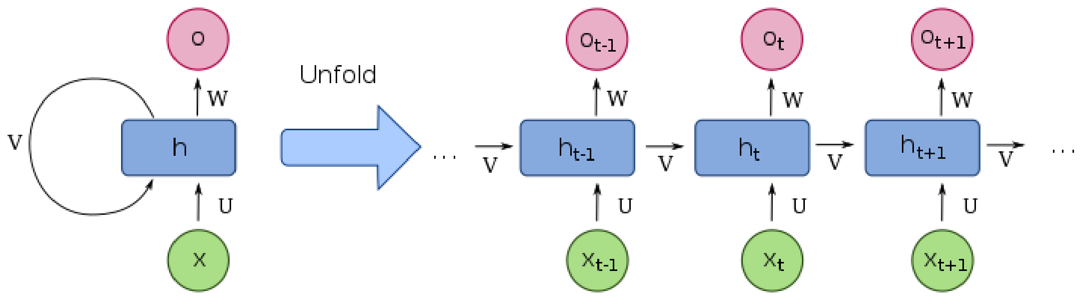


Figure 3.3: Recurrent Neural Network graphical representation (taken from [4]).

Every RNN cell computes a hidden state h_t for each input x_t, at each time-step t. The hidden state h_t is then passed to the next cell, which will process it with the input x_{t+1} and compute the next hidden state h_{t+1}. The process repeats itself until the network reaches its end. Moreover, there are three matrices that parametrize a RNN model – $\boldsymbol{U}$, $\boldsymbol{W}$ and $\boldsymbol{V}$ – which are, for every cell, responsible for a linear transformation of the inputs, states and outputs, respectively. The most basic type of RNN cell – the Elman RNN cell – introduced by [47], can be mathematically described by

$$h_t = \sigma\left(\boldsymbol{U}x_t + \boldsymbol{V}h_{t-1} + b\right) \tag{3.12}$$

and

$$o_t = \boldsymbol{W}h_t + c. \tag{3.13}$$

Where the trainable weights are the ones that compose the matrices – $\boldsymbol{U}$, $\boldsymbol{W}$ and $\boldsymbol{V}$ –, h_{t-1} is the hidden state passed from the previous cell, σ is an activation function (nonlinear) that is used to calculate the cell state h_t, x_t is the cell input and b is a bias term. For the output of the cell o_t, a bias term c is also added to the computed cell state h_t, after parameterized by $\boldsymbol{V}$. A schematic representation of a basic RNN cell, as the one just described, can be seen in Figure 3.4 where Hyperbolic Tangent (Tanh) is used as the activation function.

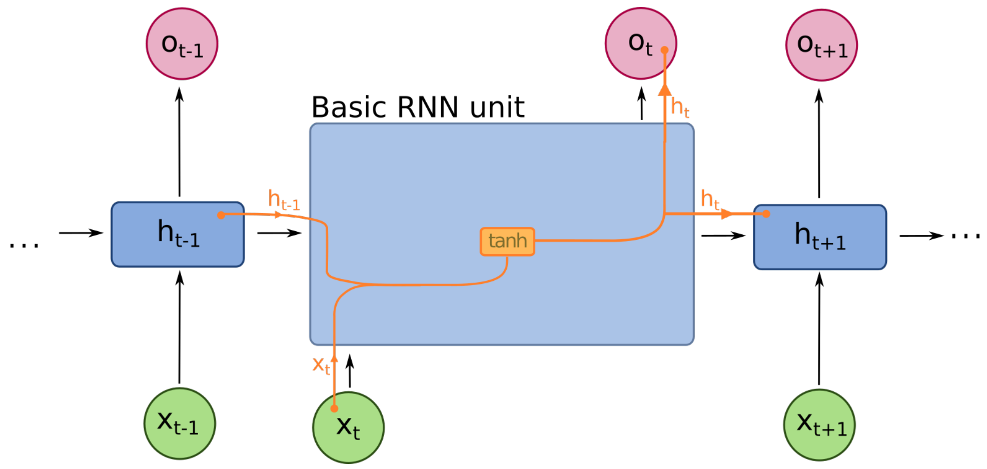


Figure 3.4: Basic RNN cell (adapted from [4]).

One limitation of the RNN architectures, is that they only flow information from the past, i.e. every cell of the network only receives information calculated by the cells behind it – it does not take into account the future cell states nor the future inputs of the input sequence. An upgraded architecture – BRNN – came to solve this problem. However, for the reasons pointed in chapter 2, regarding the application of BRNN to time-series problems (which are the focus of this dissertation), this architecture will not be covered.

Two problems observed by the basic RNN cells are the so-called vanishing gradient and exploding gradient. The training method of the RNN architecture is based on backpropagation through time, where the gradients accumulate from the output of the last cell all the way back through the entire network. Consequently, the gradients can either explode, turning the training process unstable or, become vanishingly small and get to a point where the weight values are so small that they can not decrease anymore and the training process stagnates. Therefore, the basic RNN cells are not well suited to learn long-term dependencies. The exploding gradient can be fixed with a gradient clipping approach, which basically limits any gradient from having norm greater than a defined threshold and thus, the gradients are “clipped”. On the other hand, the vanishing gradient problem would require structural modifications to be overcome. The two most commnon alternatives to the basic RNN cell that were designed to solve the vanishing gradient are the LSTM cell and the GRU cell.

3.2.3 Long Short Term Memory

The LSTM cell – proposed by [48] – was specially designed to overcome the vanishing gradient problem and learn long-term dependencies in RNN architectures. The architecture of the LSTM cell is schematically represented in Figure 3.5 where $\boldsymbol{\sigma}$ represents a sigmoid function.

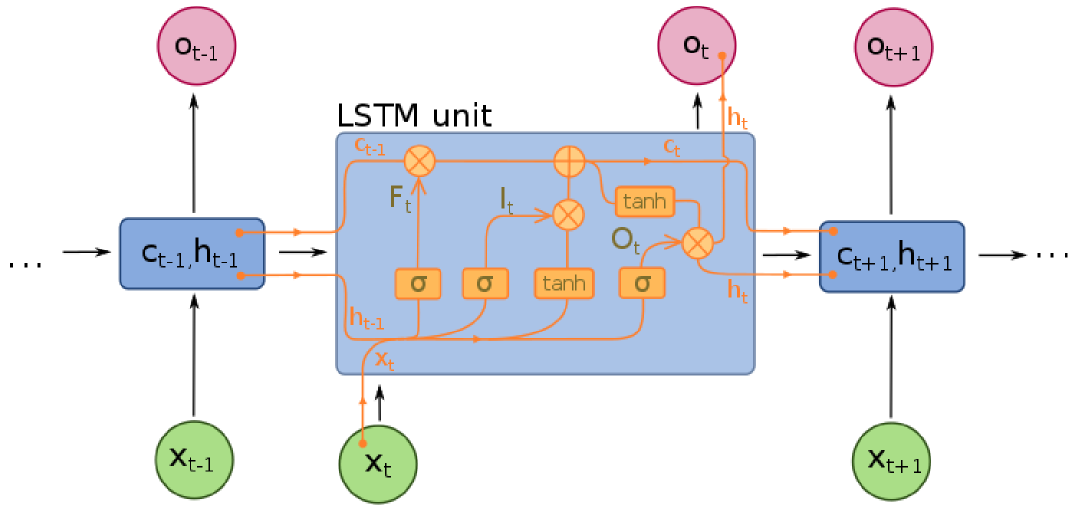


Figure 3.5: Long Short Term Memory cell (taken from [4]).

The core idea behind the LSTM is the cell state c_t and its various gates – the forget gate $\boldsymbol{F_t}$, the input gate $\boldsymbol{I_t}$ and the output gate $\boldsymbol{O_t}$ of Figure 3.5. The cell state is capable of carrying relevant information throughout the processing of an entire input sequence. This way, in the LSTM architecture the cell states are responsible for the long-term memory of the network while the hidden states are responsible for the short-term memory.

While the cell state passes on from LSTM cell to LSTM cell, information is added or removed from it by the gates. The gates are like neural networks themselves that decide which is information is allowed in the cell. As such, they are entitled of their own weight matrices – $\boldsymbol{U}$ and $\boldsymbol{W}$ for each gate – and bias terms b. With the training, the gates can learn what information is relevant to keep or forget.

3.2.3.A Forget Gate

The forget gate is mathematically described by

$$f_t = Sigmoid\left(U_f x_t + W_f h_{t-1} + b_f\right). \tag{3.14}$$

The forget gate is designed to decide what information should be stored or forgotten. It receives the information from the previous hidden state h_{t-1} and the input x_t and passes them through a sigmoid activation function. The sigmoid function computes values between zero and one, which is ideal for this purpose. If the output is closer to zero, it means the information is to be forgotten and closer to one means the information should be kept.

3.2.3.B Input Gate

The input gate is mathematically described by

$$i_t = Sigmoid\left(U_i x_t + W_i h_{t-1} + b_i\right). \tag{3.15}$$

The input gate is designed to update the cell state with new information. It also passes the previous hidden state h_{t-1} and the input x_t through a sigmoid activation function. The result will act as a filter to decide which information of the cell state will be updated. For the same reason as in the forget gate, the sigmoid is well suited for this purpose. The computation performed in Equation (3.17) is the same performed by the basic RNN cells and results in a vector of new candidates to feed the cell state. The difference is that now these candidates will be filtered by the input gate, and thus, the network has the ability to choose which information is relevant to store in the cell state (long-term memory).

3.2.3.C Cell State

The cell state is updated every time-step t, as mathematically described in

$$c_t = f_t \times c_{t-1} + i_t \times \hat{c}, \tag{3.16}$$

where

$$\hat{c}_t = \tanh\left(U_c x_t + W_c h_{t-1} + b_c\right). \tag{3.17}$$

First, it is selected what prevails from the previous cell state c_{t-1}, according to the forget gate output – the previous cell state is pointwise multiplied by the forget vector f_t. Then, the relevant information of $\hat{c}_t$ is added up to the cell state. The input gate's output i_t decides what is relevant form $\hat{c}_t$.

3.2.3.D Output Gate

The output gate is mathematically described by

$$o_t = Sigmoid\left(U_o x_t + W_o h_{t-1} + b_o\right), \tag{3.18}$$

and the hidden state passed on to the next cell is computed with

$$h_t = o_t \times \tanh\left(c_t\right). \tag{3.19}$$

The output gate controls what the hidden state h_t passed to the next cell should be. Similarly to the other gates, it receives the information from the previous hidden state h_{t-1} and the input x_t and passes them through a sigmoid activation function. Again, this output will act as a filter to decide what information from the current cell state c_t should be carried by the hidden state. Before passing through the output gate, the new cell state is transformed by another Tanh activation function.

3.2.4 Gated Recurrent Unit

The GRU cell – proposed by [49] – is the most common variation of the LSTM cell which, beyond being simpler (computationally cheaper), has proven to give similar results in many cases, as concluded in Chapter 2. The GRU also comes as an alternative to the basic RNN cells to overcome the vanishing gradient problem and learn long-term dependencies. The architecture of the GRU cell is schematically represented in Figure 3.6 where σ represents a sigmoid function.

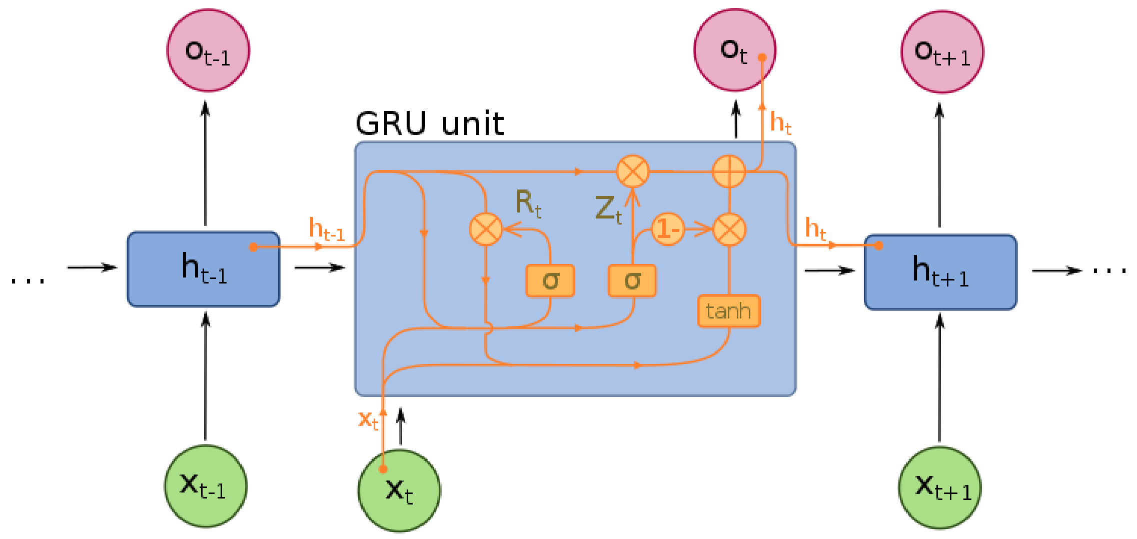


Figure 3.6: Gated Recurrent Unit cell (taken from [4]).

The main difference of the GRU cell when comparing to the LSTM is that the cell state is merged with the hidden state and thus, the GRU only comprises one state h_t. It now only has two gates – the update gate $\boldsymbol{Z}_t$ and the reset gate $\boldsymbol{R}_t$ of Figure 3.6.

3.2.4.A Update gate

The update gate is mathematically described by

$$z_t = Sigmoid\left(U_z x_t + W_z h_{t-1} + b_z\right). \tag{3.20}$$

The update gate acts similarly to both the input and forget gates of the LSTM – it decides what information to throw away and what new information to add. It receives the information from the previous hidden state h_{t-1} and the input x_t and passes them through a sigmoid activation function. Again, if the output is closer to zero, it means the hidden state information is to be forgotten. Consequently, the new computed information will be added, as it will be passed through the filter $(1 - z_t)$. And vice-versa when the sigmoid's output is closer to one.

3.2.4.B Reset gate

The reset gate is mathematically described by

$$r_t = Sigmoid\left(U_r x_t + W_r h_{t-1} + b_r\right). \tag{3.21}$$

The reset gate controls the information from the previous cell that will be used to calculate the new state. The reset gate vector r_t filters the previous input state h_{t-1} and forwards it to the calculation of the new hidden state h_t.

3.2.4.C Hidden state

The new hidden state h_t of the GRU cell is calculated as mathematically described in

$$h_t = (1 - z_t) \times \hat{h}_t + z_t \times h_{t-1}, \tag{3.22}$$

where

$$\hat{h}_t = \tanh\left(U_h x_t + W_h\left(r_t \times h_{t-1}\right) + b_h\right). \tag{3.23}$$

The new hidden state is calculated, similarly to all RNN cells, by applying an activation function (Tanh in this case) to the parameterized (by the weight matrices) input x_t and previous hidden state h_{t-1}. However, in the GRU cell, this previous hidden state already comes filtered by the reset gate r_t. This process is defined in Equation (3.23). The previous hidden stated is then passed through the update filter z_t and added to the output of Tanh after being passed by the "inverted" update filter $(1 - z_t)$. This double usage of the update filter aims to add the same "amount" of the new state that it disposed from the previous state.

3.3 Sequence-to-sequence models

Sequence-to-sequence models (also known as Seq2seq or S2S) – introduced by I. Sutskever, O. Vinyals, and Q. V. Le [53] – are a family of Machine Learning approaches designed to turn one sequence into another sequence. As concluded in Chapter 2, most of these models were designed for NLP tasks and thus, look at the inputs as words/sentences. However, they incorporate powerful properties that may be advantageous when dealing with time-series, specifically for multistep-ahead predictions. Vanilla RNN architectures compute an output as a solid unit – for multistep-ahead predictions, this unit will be a vector that symbolizes the sequential set of outputs. On the other hand, sequence-to-sequence models compute the outputs as an actual sequence, step by step, and are thus more adequate for these types of problems. The first and most basic sequence-to-sequence model is the encoder-decoder, and all the others are based on it.

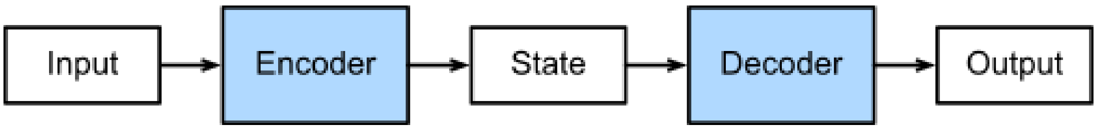


Figure 3.7: Sequence-to-sequence basis architecture (taken from [5]).

3.3.1 Encoder-decoder

The encoder-decoder model comprises two separate full RNN architectures and, as the name implies, one of them is meant to read the input data and encode it. The other is meant to decode this encoded information and transform it into a meaningful sequence. In Figure 3.8 is an example of an encoder-decoder architecture.

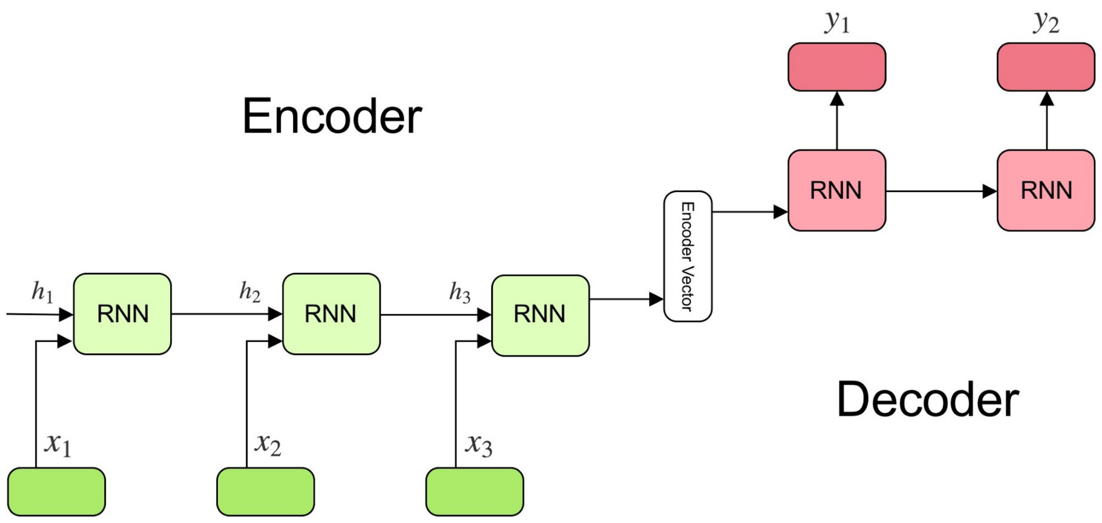


Figure 3.8: RNN encoder-decoder model example (taken from [6]).

Where the encoder is fed with a sequence of three inputs $x_t = \{x_1, x_2, x_3\}$ with length $T = 3$, and the decoder outputs the next two time-steps in the future, the sequence $y_t = \{y_1, y_2\}$ with length $T' = 2$. The encoded vector is the hidden state computed at the last RNN cell from the encoder RNN model, and it contains all the information retrieved from the input sequence. The encoded vector can also be denominated by context vector, as ti carries the context of the past values needed to model the future ones. Since the decoder is only operating in the “future domain”, its RNN cells can not be fed with input occurrences/real values. A common approach to feed the decoder RNN cells is to “reuse” the output of the previous cell, given that, if the predictions are accurate this value would be the actual occurrence to input the cell. The generalized architecture of the encoder-decoder model can be schematically represented by Figure 3.9.

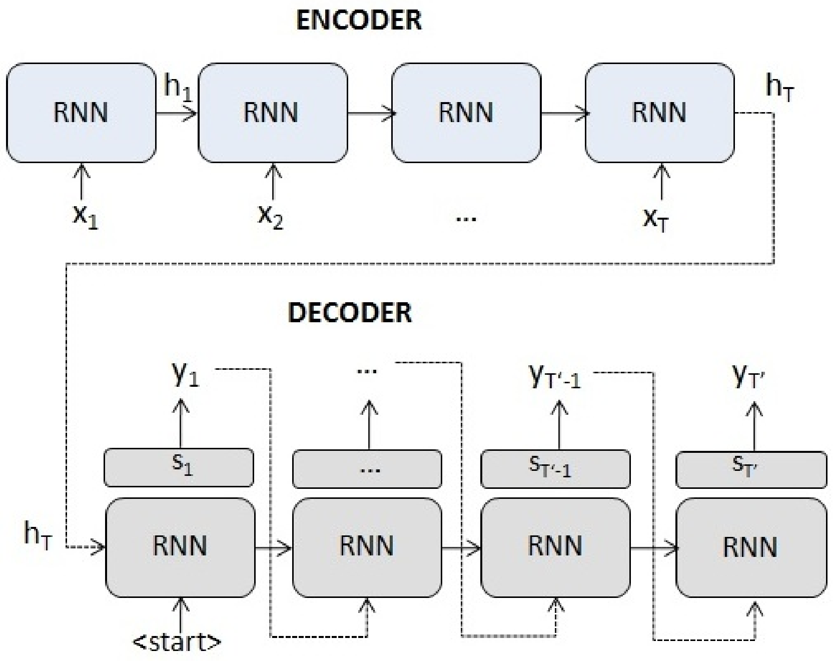


Figure 3.9: RNN encoder-decoder architecture (taken from [7]).

Accordingly, the RNN encoder-decoder model can be described by

$$h_t = f_{enc}\left(h_{t-1}, x_t\right) \tag{3.24}$$

and

$$y_t = f_{dec}\left(s_{t-1}, y_{t-1}\right), s_0 = h_T \tag{3.25}$$

where f_{enc} and f_{dec} are the RNN cells functions of the encoder and decoder, respectively, covered by Equation (3.12). x_t is the input sequence of length T and h_t is the hidden state of the encoder at time t. Also s_t is the hidden state of the decoder RNN model, and the initial state s_0 is the context vector outputted by the encoder.

Now, given that all the information processed from the input sequence has to be compressed, by the encoder, into one single vector (the context vector h_T), some information might be lost, mainly from the first elements of the sequence. This way, and in accordance with the conclusions taken in Chapter 2 regarding this subject, as the input sequences grow in length, the performance of the encoder-decoder models tends to degrade significantly. Despite less significant, the increased length of the output sequence also degrades the model's accuracy.

3.3.2 Attention mechanism

With the intent of solving the encoder-decoder liability with long sequences, the attention mechanism was developed by D. Bahdanau, K. Cho, and Y. Bengio [56]. As the name implies, this mechanism aims to give the decoder the capability to select from which outputs of the encoder RNN cells it wants to pay more attention to (assign a higher weight to). With the attention mechanism, instead of compressing the encoded inputs into a fixed-length vector, the encoder outputs a set of vectors (one for each RNN cell computed hidden state), from which the decoder will "decide", for every prediction step, which relevance to give to each of them. In is a schematic representation of the attention mechanism, with an input sequence and output sequence of lengths $T = 3$ and $T' = 3$, respectively. .

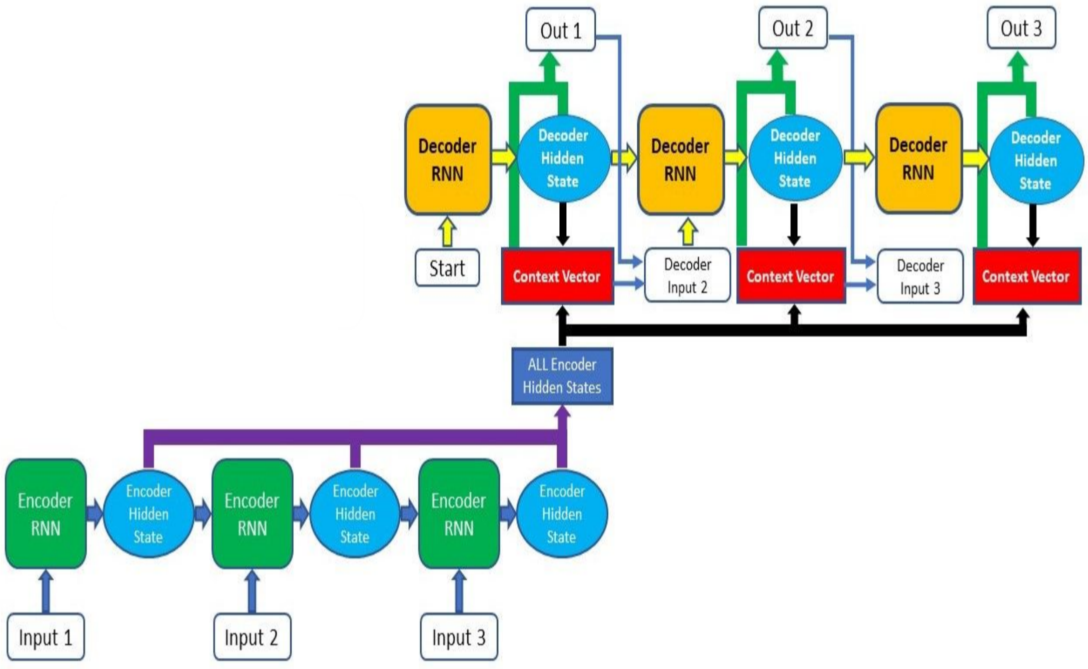


Figure 3.10: Encoder-decoder architecture with the attention mechanism (taken from [8]).

Having all the encoder RNN cells hidden states to learn from, the decoder will compute a softmax distribution, for each decoder RNN cell, to decide how much weight will be attributed to each hidden state came from the encoder. The softmax formula is mathematically described by

$$Softmax(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{|\boldsymbol{Z}|} e^{z_j}}, \tag{3.26}$$

where $\boldsymbol{Z} = \{z_1, z_2, ...\}$ is a vector with $|\boldsymbol{Z}|$ elements, in this case each element referent to a RNN cell from the decoder. The input values of the softmax function can be any real number – positive, negative or zero. However, the softamx function will transform them into values from zero to one so that they can be interpreted and probabilities. If one of the inputs is small (or more negative), the softmax will transform it into a small probability and vice-versa for a large input. Given that the outputs are meant to be interpreted as probabilities, all the outputs of softmax sum to one.

This way, during the network training, the decoder cells will learn a softmax distribution that dictates how much relevance to give to each hidden state from the encoder, having this way one different context vector for each decoder RNN cell, tuned accordingly to its needs.

In accordance with the conclusions taken in chapter 2, for long input sequences the ideal output of the softmax distribution would be very sparse and possibly with zero values. Not only the softmax function does not lead to a sparse output, but it can never output zero values. The sparsemax activation function came as an alternative [58], which is similar to the softmax function, but able to output sparse probabilities and zero values. With the sparsemax activation function the attention mechanism can be more selective and compact.

3.3.3 The Transformer

All the previously covered RNN based architectures share a computational limitation – they are not parallelizable. The recurrent mechanisms imply sequential computations, i.e. each RNN cell can only perform its computations after the previous one finishes (and passes on the hidden state). This limitation can result in having a model that is tremendously slow to train and computing predictions. To address this weakness a new simple network architecture was proposed by [9] – the Transformer – based solely on attention mechanisms but entirely without recurrence, which makes it highly parallelizable, boosting significantly the computational speed of the model.

Again, as the vast majority of sequence-to-sequence models, the Transformer was designed for NLP tasks, with a high focus on language translation problems. Therefore, the original architecture proposed by [9] integrates some modules that were placed to deal with words or word embeddings and are not applicable to time-series data. Accordingly, some posterior arrangements will be needed to have the model suitable for time-series problems. Such modifications are addressed in Section 3.3.3.D, after covering the original model for NLP. The architecture of the Transformer is represented in Figure 3.11.

The Transformer is a sequence-to-sequence model, and thus, its structure is also based on an encoder-decoder architecture. In Figure 3.11, the encoder is the block on the left, and the decoder

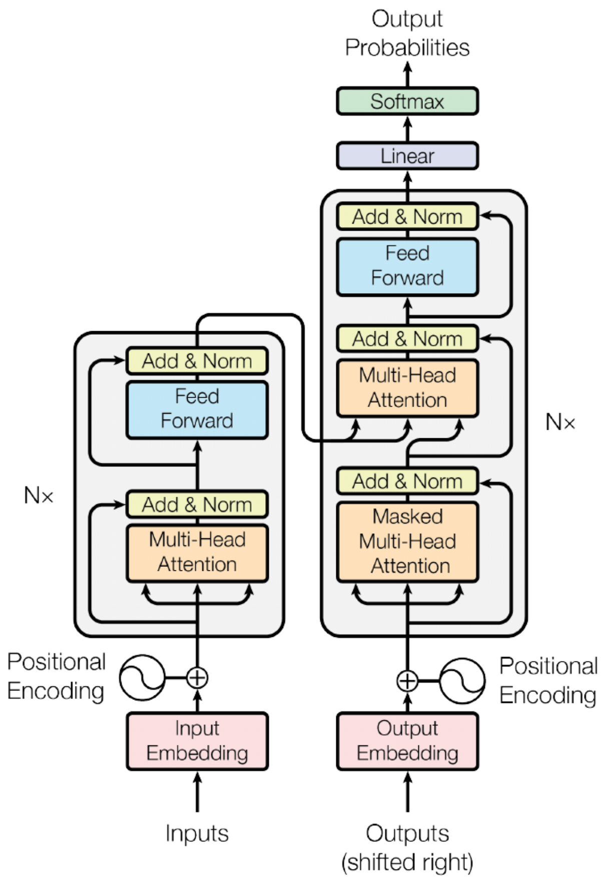


Figure 3.11: The Transformer – model architecture (taken from [9]).

is the block on the right. Both the encoder and decoder are composed of modules that can be repeatedly stacked up on top of each other, N times (represented by the Nx in the figure). This way, the encoder is formed by a set of encoding layers whose role is to map an input sequence $X = \{x_{1,2}, ..., x_n\}$ to a sequence of continuous representations $Z = \{z_1, z_2, ..., z_n\}$. These continuous representations Z are meant to determine how much related is each input with the others. The decoder, on the other hand, is composed of a set of decoding layers that, given Z, generates the intended output sequence $\{y_1, y_2, ..., y_m\}$, by computing one value at the time and integrating the already calculated values in the next predictions computations.

Encoder – In the model proposed by [9], 6 stacked encoder layers were used, $N = 6$. Each layer of the encoder comprises two sub-layers – a multi-head self-attention mechanism, followed by a fully connected FNN, placed to learn more encoding information. The Multi-head self-attention mechanism is covered in Section 3.3.3.C. Moreover, after each sub-layer is placed a residual connection [71] followed by a layer normalization [72], which translates in

$$f(x) = LayerNorm(x + Sublayer(x)), \tag{3.27}$$

where $Sublayer(x)$ is the sub-layer output.

The residual learning process essentially creates a shortcut connection between the outputs of the sub-layer and its inputs with the intent of deciding if the sub-layer computation is beneficial to the process and, if not, it will be skipped. The layer normalization function acts as a stabilizer of the learning procedure.

Decoder – In the model proposed by [9], 6 stacked decoder layers were used, $N = 6$. Each layer of the decoder also comprises a multi-head self-attention mechanism followed by a fully connected FNN, similarly to the encoder, but it integrates a third sub-layer – the masked multi-head attention – which essentially acts equally to the multi-head attention mechanism, but gets its inputs from the outputs of the decoder, sequentially, as they are calculated (without considering future decoding outputs). Moreover, the decoder's multi-head attention module takes as input both the encoded representations computed by the masked multi-head attention and the encoded representations Z from the encoder. After each sub-layer, similarly to the encoder, there is a residual connection [71] followed by a layer normalization – Equation (3.27).

3.3.3.A Positional Encoding

An important module of the Transformer architecture is the positional encoding of the input sequence, present at the bottom of both the encoder and decoder stacks. Since there are no recurrent mechanisms to explain the sequential nature of the input, the architecture needs an extra component to give some sense of order to the inputs. To serve this purpose, the positional encoding module injects information about the relative or absolute position of the different points in the sequence, i.e. it helps to determine the distance between points and the position of the points overall in the sequence.

To compute the positional encoding vectors, several functions can be used [73]. However, in [9] the authors tested the most common ones in the Transformer architecture and found similar results for all of them. In the proposed model they used sine and cosine functions of different frequencies, as mathematically described by

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right) \tag{3.28}$$

and

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right), \tag{3.29}$$

where pos is the position of an element in the sequence, i is the dimension of the element, and d_{model} is the dimension of the outputs in the output sequence of the Transformer. This way, for every element in the input sequence, there will be a positional encoding which will be different from all the others.

3.3.3.B Scaled Dot-Product Attention

The scaled dot-product attention is the central mechanism of the multi-head attention block, and thus, should be covered first. First of all, every input of the input sequence will be transformed into three vectors – query Q, key K and value V – which will be obtained by multiplying the inputs by the matrices – W^Q, W^K and W^V, respectively. These matrices are composed of parameters that will be learned when training the model. The scaled dot-product attention is composed of the operations represented in Figure 3.12.

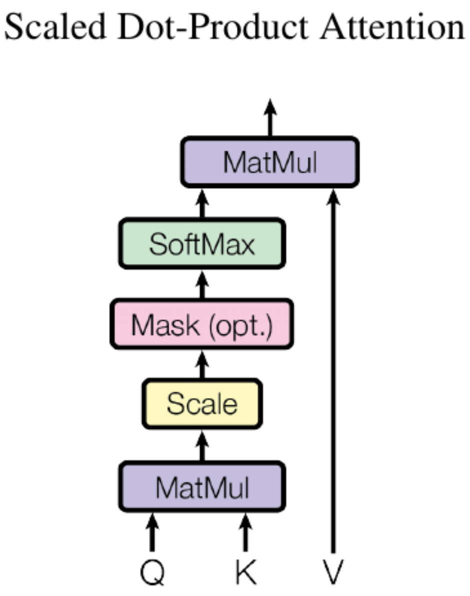


Figure 3.12: Scaled Dot-Product Attention (taken from [9]).

Mathematically, the scaled dot-product attention mechanism can be described by

$$Attention(Q, K, V) = Softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V. \tag{3.30}$$

The scaled dot-product attention is used to encode each input of the input sequence, taking into account all the other inputs and how much they are related. To do so, each query vector Q will be put against every key vector K by performing a dot-product operation which will result in a score. This score represents how much each key vector K is related to every query Q (how inputs are related to each other). These scores are scaled[1] by a factor of $\frac{1}{d_k}$, where d_k is the dimension of the key vectors (and likewise the query vectors). A softmax function is then applied to the scores to ensure that they are bounded.

All the value vectors V will then make use of the n computed scores to generate n new value vectors V', by a matrix multiplication. These new value vectors will now dictate how much influenced each input is by the other inputs – for example, if a certain input is not much relevant to another, the respective score will be minimal, and consequently, the correspondent new value vector will be reduced.

At last, all the n multi-dimensional new value vectors V' will be grouped (according to the original value vector V) and summed, which results in the output of the scaled dot-product attention mechanism.

3.3.3.C Multi-Head Attention

The multi-head attention of the Transformer is an expansion of the scaled dot-product attention mechanism and it is represented in Figure 3.13.

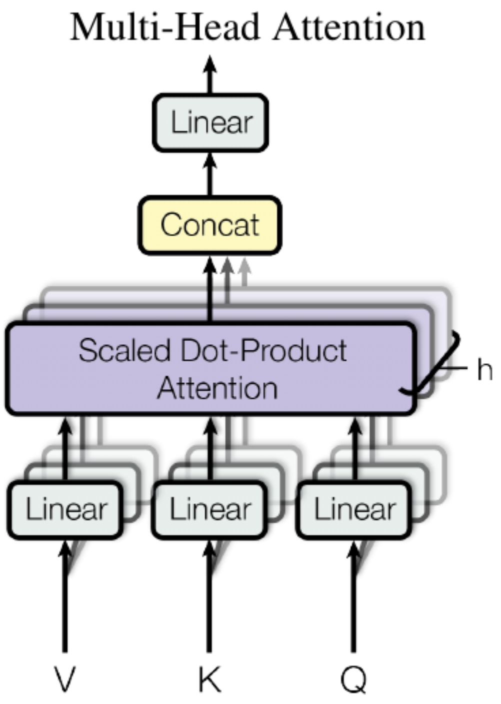


Figure 3.13: Multi-head Attention (taken from [9]).

The multi-head attention modules brings two major enhancements to the model:

[1]The scaling $\frac{1}{d_k}$ is applied to reduce the magnitude of the dot products and stabilize gradients [74].

- The capability of having every point of the input sequence taking into account all the others (self-attention), which improves the capture of context;
- The creation of different representations of each input from the input sequence, achieved by projecting them into different sets of sub-spaces.

The multi-head attention mechanism can be mathematically described by

$$MultiHead(Q, K, V) = Concat\left(head_1, head_2, ..., head_h\right) W^O, \tag{3.31}$$

where

$$head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right). \tag{3.32}$$

Fundamentally, the Transformer architecture comprises h parallel attention heads, with every each of them performing scaled dot-product attention operations. All the attention head results are concatenated and multiplied by another matrix W^O, also composed of parameters that will be learned when training the model and will, in essence, reduce the output to a single vector for each input of the input sequence. From Figure 3.13, the matrix W^O symbolizes the linear blocks – that lets the attention heads influence each other. In [9] the number of parallel attention heads used is $h = 8$. The same value was used in the implementation of the Transformer in this dissertation.

The multi-head attention blocks in the decoder still have some variants. The upper one uses the weight matrices W^K and W^V from the encoder but has its own W^Q weight matrix. For the bottom one – masked multi-head attention –, the decoder must not attend to outputs that have not been generated yet, and so, the multi-head attention block had to be modified in that regard. Otherwise, the decoder would try to learn from outputs that had not been calculated yet, and thus, could not have any meaning.

3.3.3.D Adaption to time-series problems

The Transformer original architecture is designed for NLP tasks. Opposingly to NLP, time-series problems do not work with sequences of words or characters but rather with sequences of values. Additionally, for forecasting purposes, an auto-regression process is needed and not a classification of words or characters.

Firstly, the inputs already are numerical values, as such, there should be no embeddings. Additionally, word embeddings not only have the words mapped to vectors of real numbers, but they also transform the inputs into another dimensional space. To maintain this property, a linear transformation layer can be applied to the input sequence to transform it into to a n-dimensional space. Lastly, given that, for forecasting problems, the outputs are real values and not probabilities, in the adaptation carried in this work the softmax layer at the end of the encoder was removed.

4 Implementations

Contents

4.1 Time-series Forecasting

4.1.1 Available data

A monitoring platform collected the datasets available for this dissertation from “up and running” real-world IT systems. The platform defines each metric collection as an item, and an item is mainly (i.e. relevant for this dissertation) characterized by two settings: (a) the actual metric, for example, the CPU load of computer X; (b) the collection period, for example, five minutes (5min). Every collection is then performed, for each period, only at the instant that the period clocks. For the given example, if one CPU load value is collected at 14h 00min, the next value is going to be the CPU load at the instant 14h 05min, and not the average load of the CPU between 14h 00min and 14h 05min.

The monitoring platform has two ways of storing the data regarding the items collected – history and trend – each kept separately in the database:

- **History** – keeps each collected value for a pre-defined period of time (for example, data older than one month will be discarded by a housekeeper). An example of a portion of an item’s history can be seen in Table 1.1 (with a period of 1min).

- **Trends** – basically a historical data reduction mechanism which stores minimum, maximum, average and the total number of values per every hour for numeric data types. An example of a portion of an item’s history can be seen in Table 4.1.

Table 4.1: Trend table example of CPU load [%].

Time	Minimum	Maximum	Average	Number of values
11h 00min 00s	15.32	63.35	21.33	20
12h 00min 00s	18.01	71.27	25.24	20
13h 00min 00s	15.44	54.20	20.05	20
14h 00min 00s	23.26	63.99	35.23	20
15h 00min 00s	30.98	65.83	50.50	20
16h 00min 00s	20.05	66.24	43.07	20

The idea is to keep the history as short as possible so that the database is not overloaded with lots of historical data. Instead of keeping a long history, one can keep longer data of trends (keep history for 14 days and trends for 5 years), as trends are probably enough to shape data over long horizons.

Given that the time-series forecasting study carried out in this dissertation was conducted for predictive maintenance purposes, the forecasting horizons have to be long enough so that timely preventive measures can be taken effectively. For this reason, and after an agreement with Identity on the most useful approach, the methods studied and fitted in this work were set to forecast the average value of trends for time horizons of two hours and beyond.

This way, the input data used to compute the forecasts is prevenient from trends storage, not history. Regardless, tests were also performed using data from history, but the results were always worst or similar. Given that most of the items analysed had collection periods in between one and 5 minutes, the number of steps-ahead that the intelligent models would have to predict would be too big, which could explain the worst results.

4.1.2 Data Preprocessing

4.1.2.A Normalization

Data normalization (also known as feature scaling), is a data preprocessing mechanism that is recommended and often used with ML models, especially for multivariate datasets – since the different variables/features can show very different scales, it might become hard to compare them. Since ANN models are capable of modelling any shape or function, in theory, a data normalization preprocessing step should not be required for these models to learn. However, after contracting all of the features to the same scale and around the same origin, the networks' learning parameters are less likely to explode or vanish, which will result, at least, in an improved convergence speed of the training step. Furthermore, the optimization methods used in the network's learning process have empirically shown better results with normalized data.

The most common two approaches for normalizing input datasets for ML algorithms are the standardization and the and the Min-Max scaling. The standardization method makes every variable of the dataset having a null mean and a unit variance, and it can be mathematically described by

$$f(x) = \frac{x - \mu}{\sigma}, \tag{4.1}$$

where x is the variable to normalize, μ is the mean and σ is the variance.

The Min-Max scaling method compresses every variable of the dataset into the interval $[0, 1]$, and it can be mathematically described by

$$f(x) = \frac{x - x_{min}}{x_{max} - x_{min}}, \tag{4.2}$$

where x is the variable to normalize, x_min is the minimum value of the subject variable and x_max the maximum.

There is no clear winner when choosing the data normalization method (the normalization itself does not even improve the performance that much). However, both standardization and Min-Max scaling were tested in all cases but showed similar results. The results presented in Chapter 5 were obtained with the Min-Max scaling normalization.

4.1.2.B Sliding Window

The intelligent methods studied and tested in this work are designed to receive a sequence of data as an input and output another sequence of data. For time-series forecasting, the input sequence $\boldsymbol{X}$ is composed of the last N_{in} most recent values $\boldsymbol{X} = \{X_{t-N_{in}}, X_{t-N_{in}+1}, ..., X_{t-1}, X_t\}$, where X_t represents, for an IT system MTS dataset, a vector containing each metric (feature) collected at time t. The output sequence $\boldsymbol{Y} = \{Y_{t+1}, Y_{t+2}, ..., Y_{t+N_{out}-1}, Y_{t+N_{out}}\}$ is composed of the future N_{out} values after the last input, where Y_t represents the forecasted value of the interest variable at time t.

To allow the intelligent methods to learn from historical data, they would need to be fed with a large set of input-output pairs, of the format stated in the above paragraph.The most common approach to transform a sequential time-series dataset into these input-output pairs, which is the same one used in this work, is the sliding window. This method consists in fixating a window of length $(N_{in} + N_{out})$ in the beginning of the dataset and slide it all the way until the end. Each slide constitutes one trading pair to learn from (one input sequence and one output sequence). A schematic representation of the sliding window approach is exemplified in Figure 4.1.

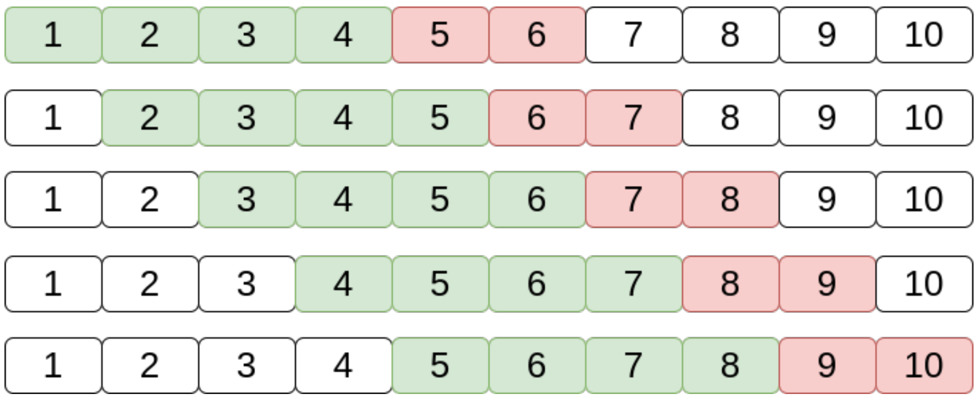


Figure 4.1: Sliding window approach.

In this example, the input sequence of length $N_{in} = 4$ is represented in green and the output sequence of length $N_{out} = 2$ is represented in red. Since the monitoring platform where the algorithms were implemented would become more versatile with variable (user-defined) forecasting horizons, a range of lengths within $[2h, 24h]$ was tested in this work for the output sequences. A vast range of input sequence lengths – $[6h, 100h]$ – was also tested to scrutinize how many steps back were needed to contain all the information necessary to perform the forecasts.

After testing all of the methods with different sequence lengths (input and output), the results showed that input sequences longer than $24h$ did not improve any of the model's performances and that for output sequences of $6h$, was length sufficient to illustrate how the models behave in multistep-ahead forecasts. For output sequences until $6h$, the forecasts did not lower the accuracy significantly. As such, the results presented in Chapter 5 for time-series forecasting were conducted with $N_{in} = 24; N_{out} = 6$.

4.1.3 Intelligent Methods Implementation

4.1.3.A Hyperparameters

The forecasting methods' hyperparameters are the configurations that are "external" to the model in the sense that they are unchangeable and defined before the training process. The ideal values change from problem to problem, so there are no pre-established ones that are universally optimal. However, to achieve a good performance with these intelligent methods, optimised hyperparameters can be crucial.

The hyperparameters optimised in the ML models applied in this are the following: (a) learning rate, (b) loss function, (c) optimisation solver, (d) activation functions, (e) number of hidden nodes and (f) number of stacked layers. For the ARIMA model, the hyperparameters to be predetermined are the constants p, q and d described in Chapter 3 – $ARIMA(p, q, d)$.

The optimal hyperparameters are frequently searched through basic approaches like brute-force or simple trial and error [75]. Furthermore, beyond the optimisation solver and the activation functions, there are automated techniques that can try to approximate the best hyperparameters, like grid search or random search.However, they can lead to inferior results and are computationally heavy [76]. The grid search was tested in this work with the ML algorithms, but the manual trial-and-error deduced parameters led to superior results.

The optimisation solver was, as well, deduced by trial and error. For all the ANN tested in this work, the optimisation solver is the one in charge of minimising the objective function of training. By far the most common optimisation solvers are: (a) the *sgdm* [77], (b) the *rmsprop* [78] and (c) *adam* [79]. The *rmsprop* and *adam* have the advantage of adapting to the used loss function. All these three optimisation solvers were thoroughly tested across all the ML models, and the *adam* showed either equal or superior behaviour in all tests. As such, the results of Chapter 5 were obtained with the *adam* optimisation solver. The loss function used in all ANN for the time-series forecasting problems was the Mean Square Error (MSE), which seems to be a common choice for this type of predictions. The loss function choice usually does not have a notable impact on a model's performance.

Finally, the learning rate used was $lr = 0.05$; tests with other values between $[0.01, 0.1]$ were experimented but did not change the results. And for last, all of the activation functions described in Section 3.2.1 were tested, one for each layer of each ANN (all combinations), but, except using only linear units that showed worst results, the performances were similar and thus, this choice is irrelevant.

4.1.3.B Cross-validation

The goal of fitting a model to a training dataset is to have the model being able to generalise its knowledge to new unknown data. However, feeding a dataset to an intelligent method to model it sometimes leads to an excessive learning of the training dataset in particular. This event is defined as over-fitting, and will result in a model that performs (excessively) well with the training dataset but will degrade unreasonably when put against data outside of the training set.

Cross-validation is a training method designed to prevent an over-fitting occurrence. It consists of splitting the dataset into K segments (folds), equal in size. Then, $K-1$ folds will be used as the training set for the model to learn from. The remaining fold will be used as the test set. This procedure is repeated, using the same folds split, but always using a different fold for the test set until all the possibilities are carried out. The iteration that showed better results with the underlying test set will be the one from which the final model will be taken. This procedure is schematically represented in Figure 4.2, with $K = 5$.

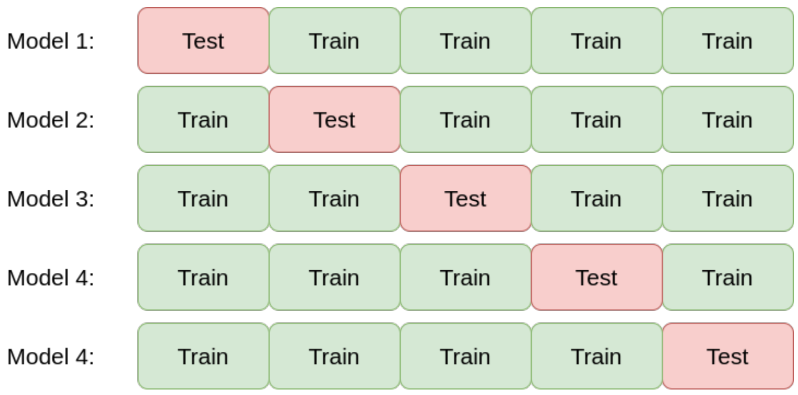


Figure 4.2: Cross-validation with $K = 5$ folds.

Furthermore, given the time ordering present in time-series, the Cross-validation method might ignore the sequential nature of time. Namely, instead of just wanting the model to generalise to unseen new data, for time-series forecasting, it is wanted that the model generalises for future data. This way, the method used to secure that the model is only tested for a data latter in time than the training set, is the one represented in Figure 4.3. It should also be noted that, for models that show weak accuracies even for the training dataset, over-fitting is not a concern and methods like cross-validation might be discarding data that would improve the models' performance. This way, when such models are put into production, these overfitting-preventive methods should be avoided.

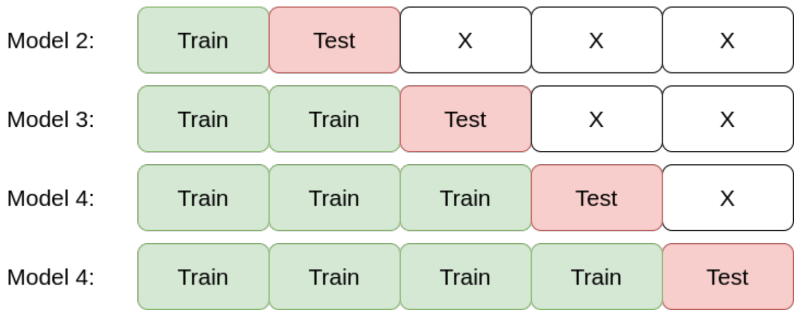


Figure 4.3: Cross-validation for time-series with $K = 5$ folds.

4.1.3.C Evaluation Metrics

The metric chose in this work to evaluate the time-series forecasting models was the MSE, that is mathematically decribed by

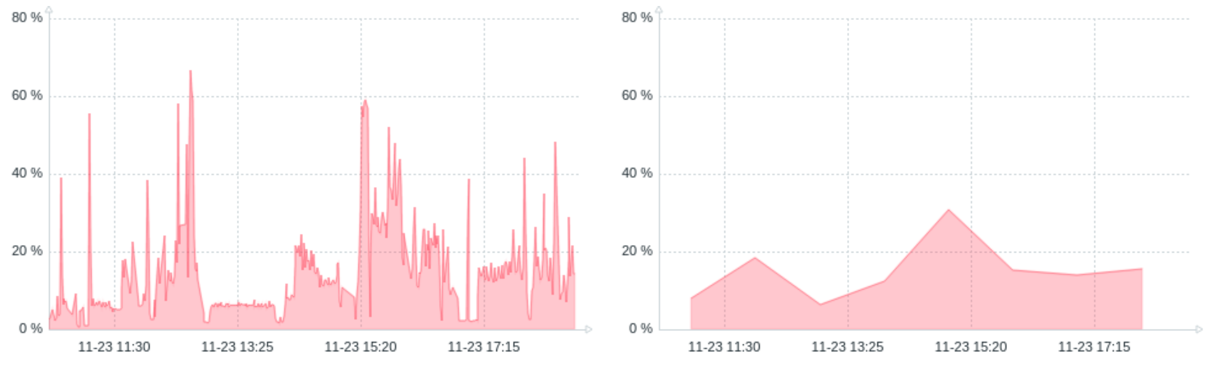
$$MSE = \frac{\sum_{i=1}^{N} \left(y^{i}_{real} - y^{i}_{forecast} \right)^{2}}{N}, \tag{4.3}$$

where y_{real} are the true values – the ones present in the test set outputs –, $y_{forecast}$ and the values forecasted by the model, and N is the length of the test set.

4.1.4 Period Reduction trial

This experiment was only tested with the ANN architectures.

Although the trend tables (one-hour averages) might be able to properly shape data over long horizons, some spontaneous incidents that only happen momentarily could also compose valuable information to compute the forecasts, as they might be related to meaningful events with long term impacts. For example, several CPU spikes during a couple of minutes could indicate the start of several different applications or services, that would increase the average CPU consumption over the next hours. In Figure 4.4 is a visual example that illustrates how the trends storage misses irregularities in data that are caught by the history storage.

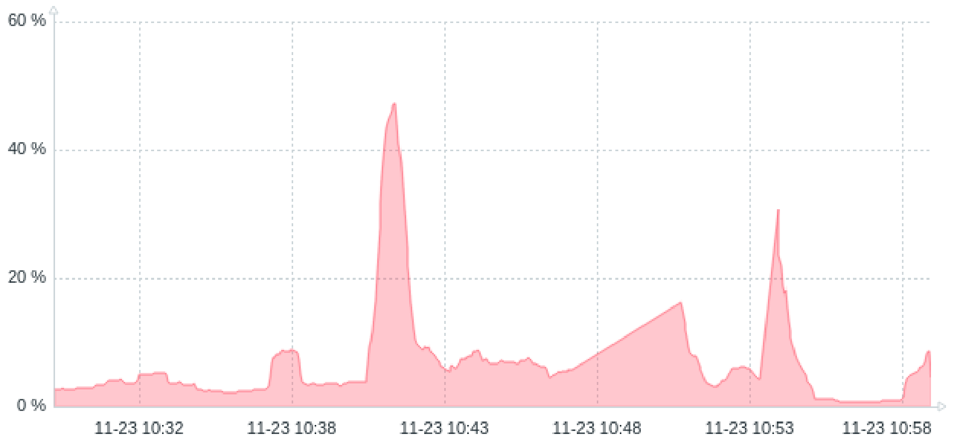


(a) CPU load – history storage.

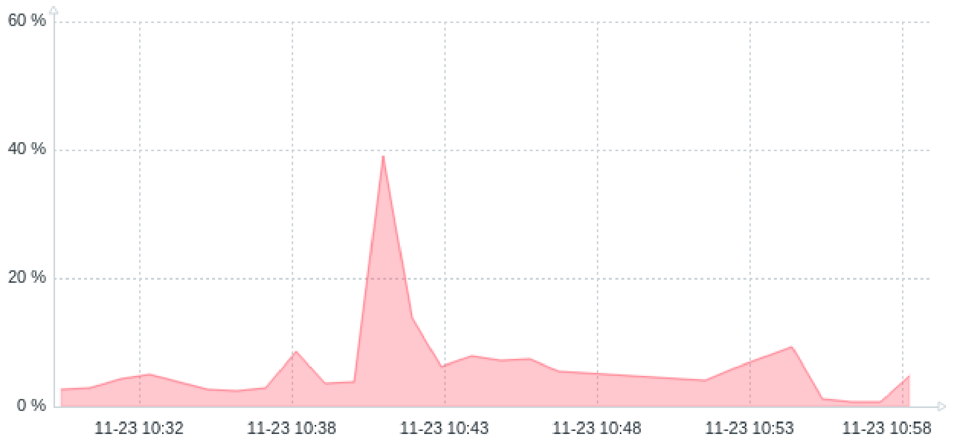
(b) CPU load – trends storage.

Figure 4.4: CPU load comparison of the two different storage types: history vs trends; the item's collection period is one minute.

Moreover, given that the metrics collection is not continuous, using the minimum period possible will increase catching spurious occurrences. To test this possibility, items were set to perform collections with the minimum period allowed by the monitoring platform, 1sec. An example of a CPU load collection comparison between a period of 1sec and a period of 1min is present in Figure 4.5.



(a) CPU load with a period collection of 1 second.



(b) CPU load with a period collection of 1 minute.

Figure 4.5: CPU load comparison of two different collection periods: 1 second vs 1 minute.

It can easily be noted that there are spikes present in Figure 4.5(a), that were not caught with a period collection of one minute – Figure 4.5(b).

It must also be noted that this history storage usage is highly incompatible with the monitoring platforms' purpose. According to the platforms' documentation, the history storage is kept as short as possible given that it consumes much more disk space then the trends storage (which is precisely the point of using trends), and thus keeping history for long periods of time, which would be needed to train the intelligent forecasting models, could be unbearable. Furthermore, the collection of data at very short periods (as one second), weights a significant burden in an IT system, as it needs to be continually polling data from the monitored metrics. A monitoring platform is integrated into an IT system because it can help to maintain it and to keep it reliable. Thus, if the monitoring tools used consume a lot of resources, they will interfere with the system and might weaken it, instead of the opposite.

This way, the experiment described in this chapter was conducted to try to understand if eventual forecasts with bad accuracies could be due to the loss of information inherent to the use of trends storage or too large collection periods.

To try to make some meaning out of the spikes noted in datasets like Figure 4.5(a), an approach that aims to count the number of spikes and feed it to the intelligent method was conducted. First of all, given the lack of available history data with short collection periods to train the models, trend tables of 10min averages were manually created to increase the number of time-steps. Using this "artificial" trend table, the same number of time-steps were used for the input and output sequences – $N_{in} = 24$ and $N_{out} = 6$. Then, instead of just using the sliding window approach, described in Section 4.1.2.B, to generate sequences to feed the model, K extra points will be added to the sequence, where K is the number of input features. Each of the K points will be the total count of spikes, $\#spikes$, of the respective feature, during the whole input window time segment. Note that each input is referent to a 10min average, while the spikes count is performed in the whole history data (one value per second). As such, for an input sequence of four time-steps, the input sequence timeline accounts for $4 \times 10 \times 60 = 2400$ seconds, which is the number of points evaluated to count the spikes.

In Figure 4.6, is an example of the upgraded sliding window just described, but with $N_{in} = 4$ and $N_{out} = 2$.

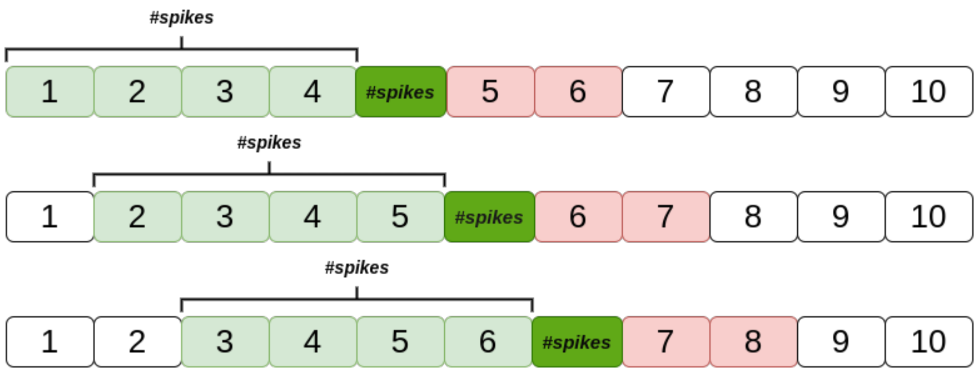


Figure 4.6: Sliding window for the period reduction trial.

The spikes count – $\#spikes$ – was calculated with a trial and error approach. Looking at each feature's graphical data, like Figure 4.5(a), define a threshold value, on top of the mean of the whole sequence, above which a spike would be considered abnormal. For example, if the mean of an input sequence is $\mu = 7$ and defined threshold is $T = 30$, for each time that the metric's collected value crosses the value $\mu + T = 37$ during the subject input sequence timeline, the spike count is incremented. For metrics like free memory, where a lower value is the "problematic" and not the other way around (like CPU load), the spike count is incremented when the line $\mu - T$ is crossed.

After several trial and error iterations, pick the threshold that led to better results and use it to the final forecasting model.

4.2 Event Prediction

4.2.1 Available Data

The available data from the monitoring platform for the event prediction problems are of two types: trends and events. The trend values are of the same type described in Section 4.1.1. The event data is directly related to events, and it comes in the form described in Section 1.2.2 (not periodical like history and trends). An example of an event table is also given in Table 1.3.

The events monitored in this work were related to problems in IT systems, in particular, services shutting down unexpectedly or becoming unavailable. This way, when the system shuts down or becomes unavailable – the event/problem is triggered – and at that instant, a row in the respective event is appended with a value of 1. When the system recovers from the problem, another row is appended with a value of 0.

The available data from trends is related to metrics collected in the same machine/IT system where the events occur.

4.2.2 Data Preprocessing

4.2.2.A Event to time-series

Given that the event points present in the datasets are not periodical, the intelligent methods studied are not prepared to receive such data as input and compute predictions with it. To convert the original event datasets to periodic sequences suitable for the ANN models, a preprocessing step that results in a binary time-series was applied:

1. Round the timestamps of the events to the closest (past) hour (because the trends have values per hour);

2. Generate a time-series with 0's as values for the same timeline and with the same period (one hour) of the trends metrics that will be used.

3. For the time between every event occurrence and its respective recovery, flip the values of the time-series to values of 1.

After applying this technique, the original events from Table 1.3 would be transformed into the time-series table in Table 4.2

Table 4.2: Time-series event dataset example.

Timestamp	Event
2020-01-05 00:00:00	0
2020-01-05 01:00:00	1
2020-01-05 02:00:20	0
2020-01-05 03:00:20	0
...	...
2020-01-08 03:00:00	0
2020-01-08 04:00:00	1
2020-01-08 05:00:20	1
2020-01-08 06:00:20	1
2020-01-08 07:00:20	1
2020-01-08 08:00:00	0
2020-01-08 09:00:00	0
...	...
2020-01-09 21:00:00	0
2020-01-09 22:00:00	1
2020-01-09 23:00:00	1
2020-01-10 00:00:00	1
2020-01-10 01:13:05	0

Now that all of the data available (metrics and events) is in a time-series format, the events can also be used as inputs, as if they are just another feature, and the ML can work just like for a time-series forecasting procedure. To predict one event, simply choose as the feature to "forecast", the respective event time-series.

4.2.2.B Normalization

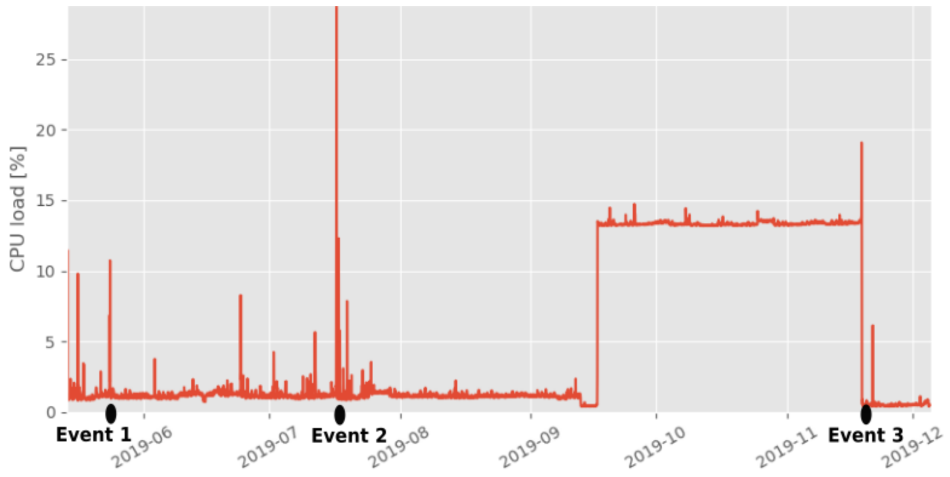
The same normalization described in Section 4.1.2.A was applied to the metrics as inputs used in the event prediction. Given that the events time-series datasets already comprises values between $[0, 1]$, there is no need to normalize them.

4.2.2.C Sliding Window

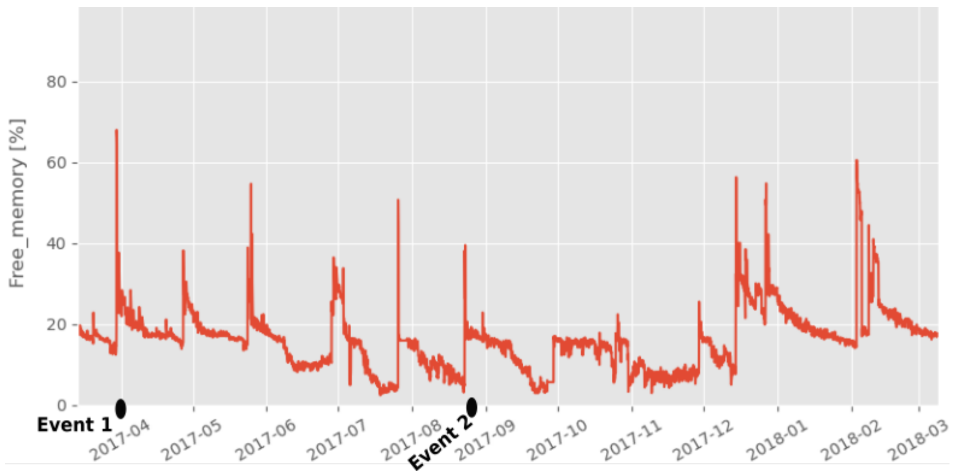
The same sliding window approach described in Section 4.1.2.B was used with the addition of the event time-series. Furthermore, since the monitoring platform where the prediction algorithms are implemented would become more versatile with variable (user-defined) event prediction horizons, a range of lengths within $[2h, 24h]$ was tested in this work for the output sequences. A vast range of input sequence lengths – $[6h, 100h]$ – was also tested to scrutinize how many steps back were needed to contain all the information necessary to perform the predictions.

After testing all of the ML architectures with the different lengths, the results showed that input sequences longer than $20h$ did not improve any of the model's performances and that for output sequences longer than $4h$, the performance started to deteriorate notably. For output sequences until $4h$, the forecasts did not lower the accuracy significantly. As such, the results presented in Chapter 5 for event prediction were conducted with $N_{in} = 20; N_{out} = 4$.

4.2.2.D Oscillations detection



(a) CPU load.



(b) Free memory.

Figure 4.7: Graphical representation of CPU load and free memory with event occurrences highlighted.

After looking at the available metrics with graphical representations and crossing them with the respective event occurrences, it could be suggested that some of the metrics revealed destabilised behaviours in the times tightly close to an event occurrence. Such occasions are exemplified in Figure 4.7.

In an attempt to extract this information from the data and feed it directly to the network, an approach using standard deviations to capture those irregularities was used (every feature separately): for each point of each input sequence, compute the standard deviation of the last K values and use it as an extra input to feed the network. This way, each point of the input sequence would now be a 2D vector with (a) the actual value of the metric at the respective time-step and (b) the standard deviation of the previous K time-steps of the subject metric.

This constant K used for the horizon of the standard deviation computations was deducted with a trial and error manual approach. The value that revealed best results was $k = 10$ and thus, is the one used for the results presented in chapter 5.

Several other methods were experimented in place of the standard deviation, like the harmonic mean and a spike count similar to Section 4.1.4, but none of them led to superior results.

4.2.2.E Minority Class Oversampling

Given that the events evaluated in this work are derived from problems and failures, such occurrences only happen sporadically. This way, as the prediction approach used computes, for the next N_{out}, if the event will be triggered or not, for the vast majority of the times, the right prediction will be "no occurrence". If the two possible outputs (0 and 1) are labelled into two classes – the minority class and the majority class –, the first would be for a prediction of 1 and the second for a prediction of 0.

The problem with such a small minority class (1) is that, if the model learns to "blindly" forecast the majority class (0) regardless of the inputs – although useless –, it will end up with a very high accuracy in the training set. This way, the model is encouraged by the data to learn this useless prediction.

To counter the majority class dominance, the minority class oversampling method was used. This method identifies the minority class occurrences in the training set and replicates them until the two classes are balanced enough for the model to learn to output both of the predictions. A value of 0.5 for the minority class oversampling technique ratio, $R_{OMC} = 0.5$, will result in a dataset where both classes have the same number of occurrence. Is this number is less than 0.5, the majority class will dominate by the correspondent ratio. If the oversampling ratio is more than 0.5, the (initially) minority class will dominate the dataset, by the respective ratio R_{OMC}.

4.2.3 Intelligent Methods Implementation

For the ML methods tested in the event prediction problems, the hyperparameter search was conducted like described in Section 4.1.3.A as well. The optimisation solver's conclusions were the same – the *adam* optimisation solver showed either equal or superior behaviour in all tests. However, since this is now a classification problem instead of a regression, the loss function used is the Binary Cross Entropy (BCE), which also seems to be a common choice for this type of predictions.

Similarly to the forecasting study, for event prediction, after several learning rates experimented between $[0.01, 0.1]$ ending up with the same results, the learning rate used in all final models was $lr = 0.05$. The activation functions, on the other hand, took a slight difference from the forecasting models – the output activation function (after the last layer) was always the sigmoid, which is the most common for problems where the output should be between $[0, 1]$. For the other activation functions, all of the ones presented in section 3.2.1 were tested, and again, the results were all very similar which makes this choice irrelevant as well.

4.2.3.A Evaluation Metrics

Again, for this type of problems, accuracy is not a good indicator of performance. A method that never predicts an event will probably have a high accuracy by merely predicting that the event never occurs.

This way, to evaluate the event prediction models, two measurements were used – precision and recall. Precision and recall can be mathematically described by

$$Precision = \frac{TP}{TP + FP} \times 100 \tag{4.4}$$

and

$$Recall = \frac{TP}{TP + FN} \times 100 \tag{4.5}$$

where TP stands for true positives, FP stands for false negatives, and FN stands for false negatives. In practical terms, the precision measures how accurate each positive event prediction is, i.e. how trustworthy is the model when it predicts that a problem will be triggered. Moreover, the recall measures how many event occurrences the model missed to predict – a model that never or too rarely predicts an event occurrence will show a recall close to zero. Both the precision and recall measurements compute values in between $[0, 100]$. The perfect model would score one at both.

5 Results and Discussion

Contents

5.1 Time-series Forecasting

The algorithms developed in this dissertation were put into production environments and tested across several different machines and with different metrics. For the results report, a set of data (collected from the same machine) was chosen to illustrate the performance of the embraced algorithms. The forecasts obtained with the chosen set of data are representative of the overall picture, in the sense that the conclusions drawn from them also apply to most of the results obtained, and thus can be generalised.

A short sample of two months of data from the metrics used in this report for the time-series forecasting study is represented in Figure 5.1. The whole dataset accounts for three years of past data, and for the results presented in this chapter, it was split into a training set and test set for 80% and 20%, respectively.

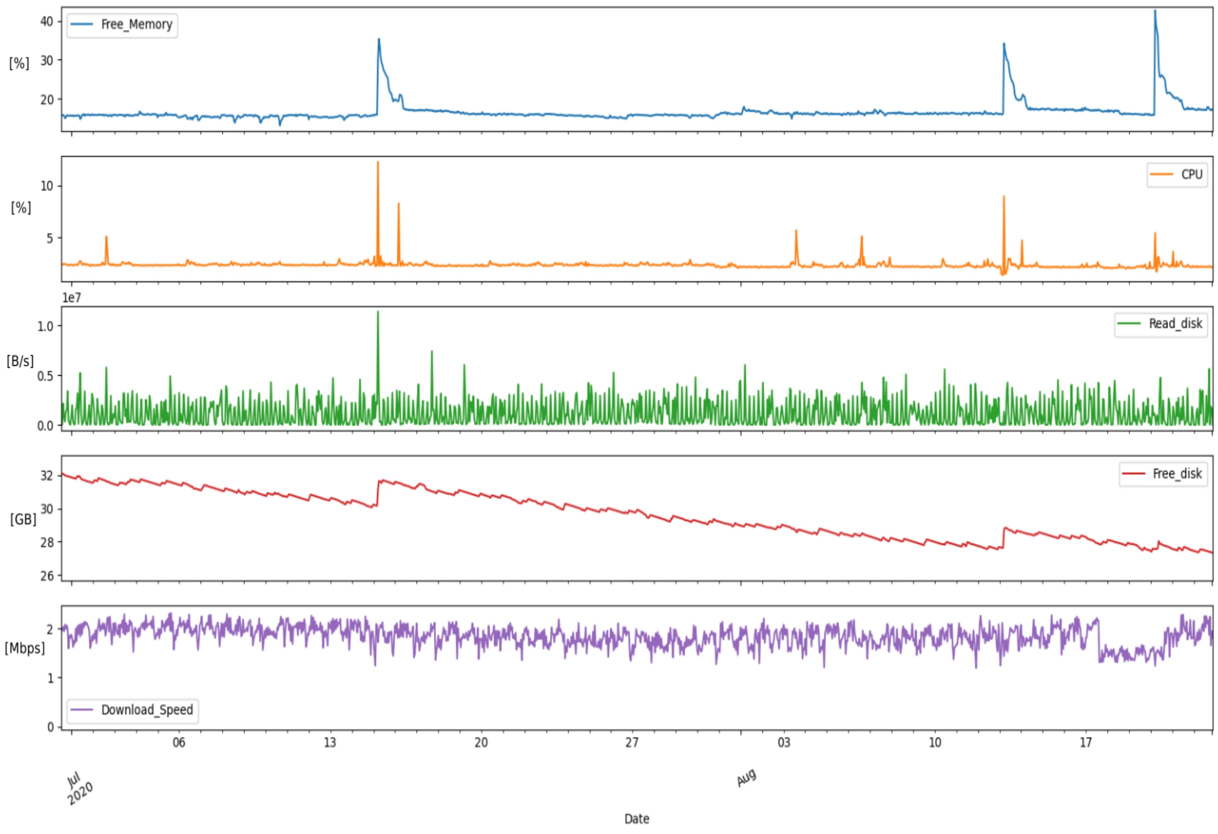


Figure 5.1: Sample of the time-series dataset used to evaluate the models.

The first metric, represented in blue, is the free RAM memory of the machine, in percentage. The second metric, represented in orange, is the CPU load of the machine, in percentage. The third metric, represented in green, is the rate of bytes per second that are being read from disk. The fourth metric, represented in red, is the free disk space, in GB. The fifth and last metric, is the download speed of the machine, in Mbps. Out of the five metrics, the two that had more interest in being forecasted (product wise) were the free RAM memory and the CPU load. The results obtained in both of the forecasts were similar. This way, even though the free RAM memory (blue), will be the forecasted and evaluated metric in this chapter, the conclusions also apply to CPU load forecasts.

All of the ANN models introduced in Chapter 3 were thoroughly tested for different numbers of hidden nodes and stacked layers. The rest of the hyperparameters are already defined in Section 4.1. For the ARIMA method, since the goal was to use it as the statistical reference to compare the ML models with, rather than optimize it thoroughly, a python module $auto\_arima()$ [76] was used to estimate the optimal parameters.

A table with the range of hidden nodes and stacked up layers tested in each model is present in Table 5.1. The iterative tests over the number of hidden nodes were carried with steps of 5 more nodes, i.e. to test the hidden nodes between $[15, 100]$, the total set of numbers tested is $\{15, 20, 25, ..., 90, 95, 100\}$.

Table 5.1: Ranges of hyperparameters experimented for the forecasting models.

Forecasting model	Hidden nodes	Stacked up layers
FNN	[15, 100]	[1, 10]
Vanilla RNN variants	[15, 500]	[1, 5]
Encoder-decoder variants	[15, 150]	[1, 5]
Attention Encoder-decoder variants	[15, 150]	[1, 5]
Transformer	8	[1, 15]

The variants are referent to the three RNN cells studied in this work – basic RNN, LSTM and GRU. The Transformer hidden nodes are referent to the parallel attention heads used in the multi-head concatenation (the same number used in the original work [9] was used).

Unfortunately, none of the models achieved satisfactory performances, and none of them accomplished forecasts good enough to be worth being used in a monitoring platform to assist in PdM. However, the configuration of each architecture that performed better (least bad), in terms of MSE, are written in Table 5.2.

Table 5.2: Best performing forecasting models and corresponding MSE evaluations.

Forecasting Model	Hidden nodes	Stacked up layers	MSE (t)	MSE (t+2)	MSE (t+5)
ARIMA	—	—	58.33	63.27	61.68
FNN	35 — 40 — 25	3	64.27	46.65	38.35
Vanilla RNN	40	1	11.53	13.12	14.70
Vanilla LSTM	50	1	6.61	9.75	11.89
Vanilla GRU	40	1	9.12	12.34	13.31
Encoder-decoder RNN	20	1	13.38	13.92	14.37
Encoder-decoder LSTM	20	1	13.47	13.91	14.73
Encoder-decoder GRU	20	1	13.64	13.25	14.86
Attention Encoder-decoder RNN	20	1	13.36	13.42	14.69
Attention Encoder-decoder LSTM	20	1	13.18	13.22	13.93
Attention Encoder-decoder GRU	20	1	13.43	13.27	14.70
Transformer	8	3	13.53	13.63	14.56
Naive approach	—	—	3.61	9.74	14.51

In accordance with the implementation described in Section 4.1, the used input sequence is of 24 time-steps and the output sequence of 6 time-steps. As the datasets are from trend tables, their collection period is of 1 hour, and the outputted forecast has an horizon of 6 hours. This way, the first (t), the middle $(t+2)$ and the last $(t+5)$ forecasts respective performances are present in Table 5.2.

Furthermore, for comparison purposes, an extra row is appended with the results of the naive approach. The naive approach is an estimating technique in which the last period's values are used "blindlessly" as the next period's forecast, without adjusting them or attempting to establish causal factors. For multistep-ahead forecasts, the naive approach can be described by

$$y_{forecast}(t+P) = y_{real}(t-1), \tag{5.1}$$

where $y_{forecast}(t+P)$ is the "computed" forecast, $y_{real}(t-1)$ is the last observed value, and P is the step count of the forecast. The naive approach will result in a delayed time shift (to the right) of the real values. A graphical representation of the forecasts obtained for a sample of the test set is present in Figure 5.2.

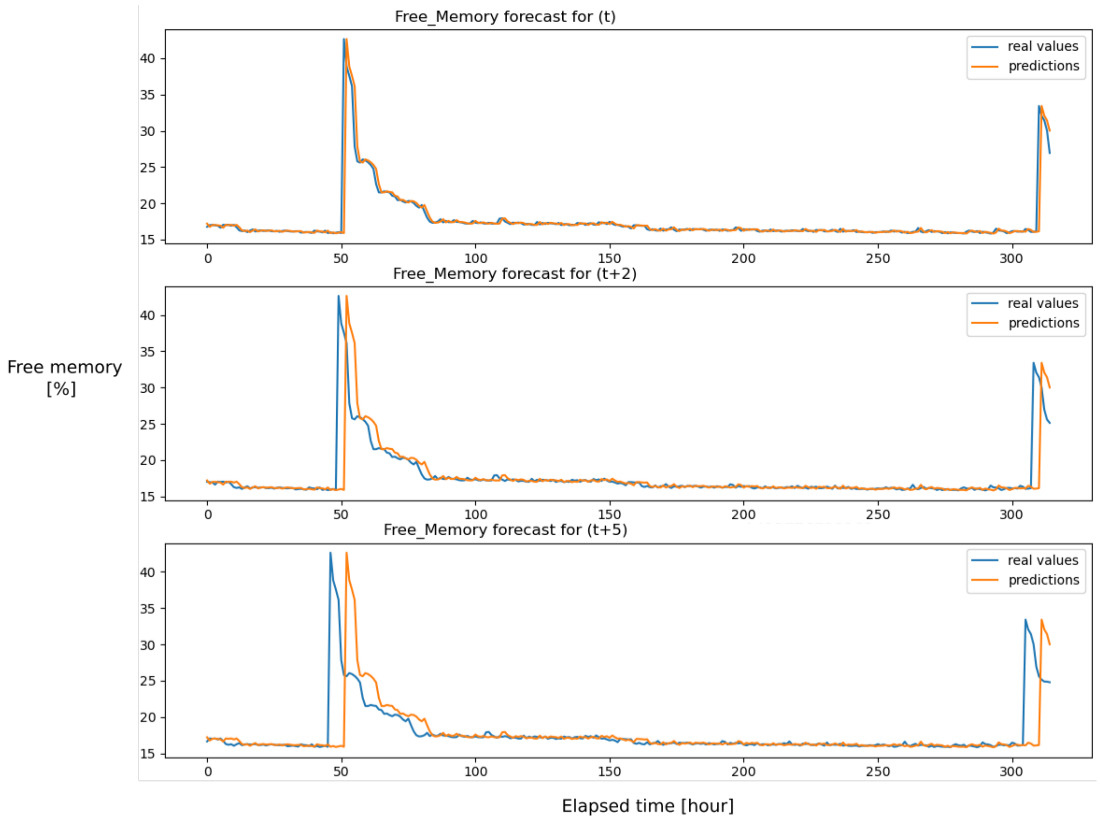


Figure 5.2: Forecasts of the naive approach

Despite the fact that, for real values that are more constant over time, the naive approach will output many forecasts with a low MSE , these forecasts are of no practical use because they will not add more information than the one already present in real-time monitoring. This way, for an intelligent model to be considered useful in this problem, its forecasts would not only have to compute forecasts with an MSE significantly below the naive approach but also detect abnormal behaviours before they occur.

Regardless of the bad results, some conclusions about what did the models learn from the data could be taken from their behaviour and graphical representations.

The “simplest” networks – the FNN – have fewer parameters to learn from and thus, in theory, are able to retain less information than the most robust ANN architectures. From their forecasts, it seems that the FNN were simply not able to model the training dataset and, as such, performed very poorly. However, it is notable that the last positions of the output sequence (the forecasts of further time-steps in the future) are less oscillatory than the first ones, which can be seen in Figure 5.3. This is the reason why in Table 5.2, the MSE of the forecast $(t+5)$ is smaller than the one of $(t+3)$.

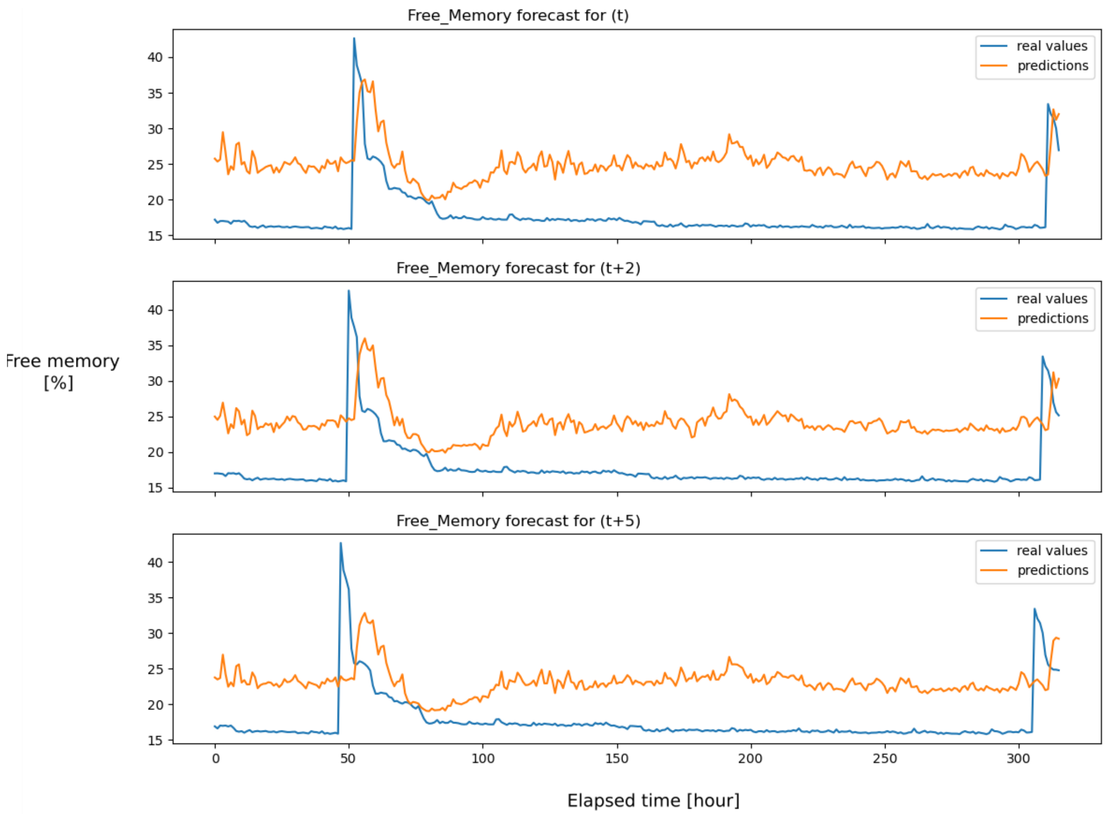


Figure 5.3: Forecasts of the FNN model with 3 layers of 35, 40 and 25 hidden nodes, respectively.

The architecture that best performed in terms of MSE was the vanilla LSTM, and it did so because it managed to learn something close to a naive approach, and not by “intelligently” computing forecasts. A sample of the forecasts computed by the “winner” vanilla LSTM is graphically represented in Figure 5.4.

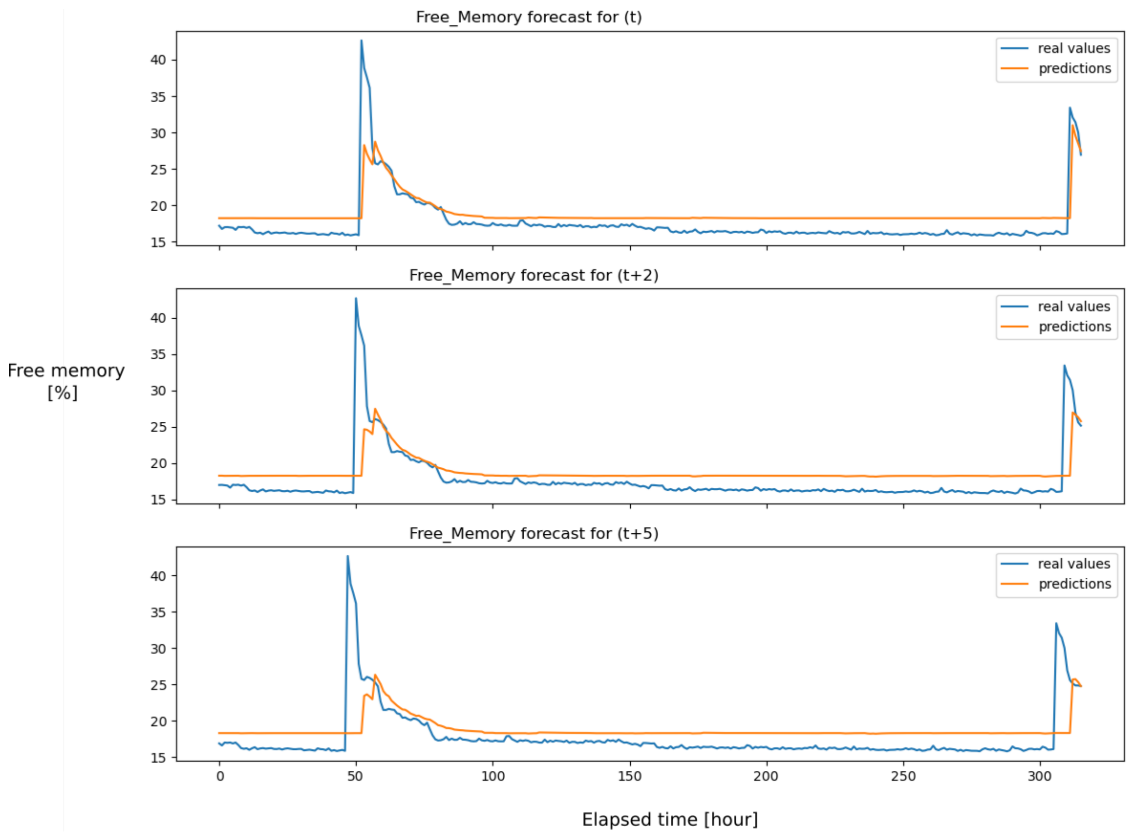


Figure 5.4: Forecasts of the vanilla LSTM model with 50 hidden nodes and 1 single layer.

It can be deducted from Figure 5.4 that the network learned to output a constant for most of the time and that when it detects a steep oscillation, it reacts with a softened naive approach – softened in the sense that, above a certain amplitude, it replicates the oscillation but with a less steep reaction.

For the same vanilla LSTM architecture with less hidden nodes (20), the forecasts are similar but much closer to the naive approach. The oscillation reaction is not so softened, and the rest of the forecast looks like a correct naive approach (but with an offset). A sample of the forecasts of this network is present in Figure 5.5.

It is possible that the LSTM is able to learn that by applying a naive approach, it will achieve a smaller MSE (which is the mathematical goal of the training process). Moreover, a LSTM network with more nodes can learn more information and is able to learn that by outputting a constant value and softening the oscillations reaction, the MSE can be even smaller.

Furthermore, the most oversized LSTM networks tested in this work went even further and learned to just output a constant regardless of the input sequence. As an example, a forecast of a LSTM network with 3 stacked up layers of 200, 500 and 50 hidden nodes, respectively, is represented in Figure 5.6(a).

Just like the biggest vanilla LSTM models, the larger and more complex architectures – Encoder-Decoder, Encoder-Decoder with Attention and the Transformer – learned to forecast a constant output

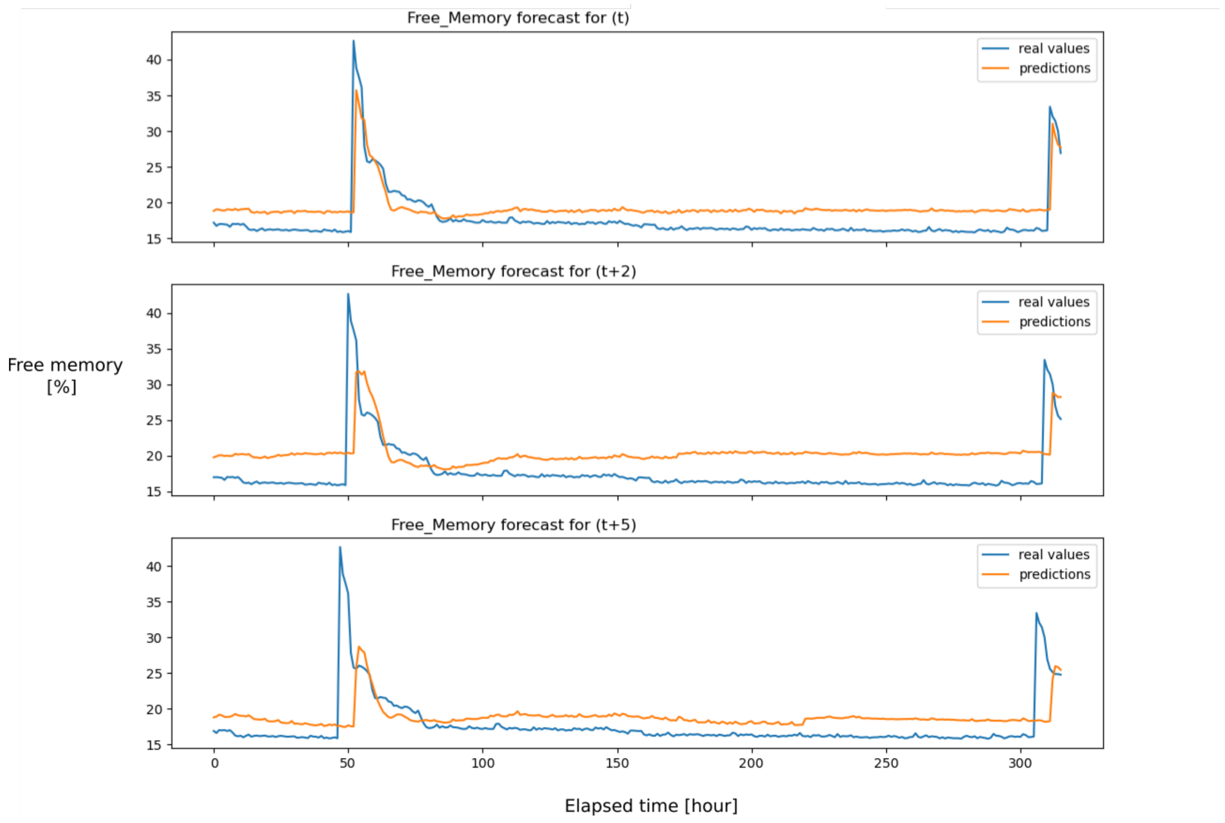
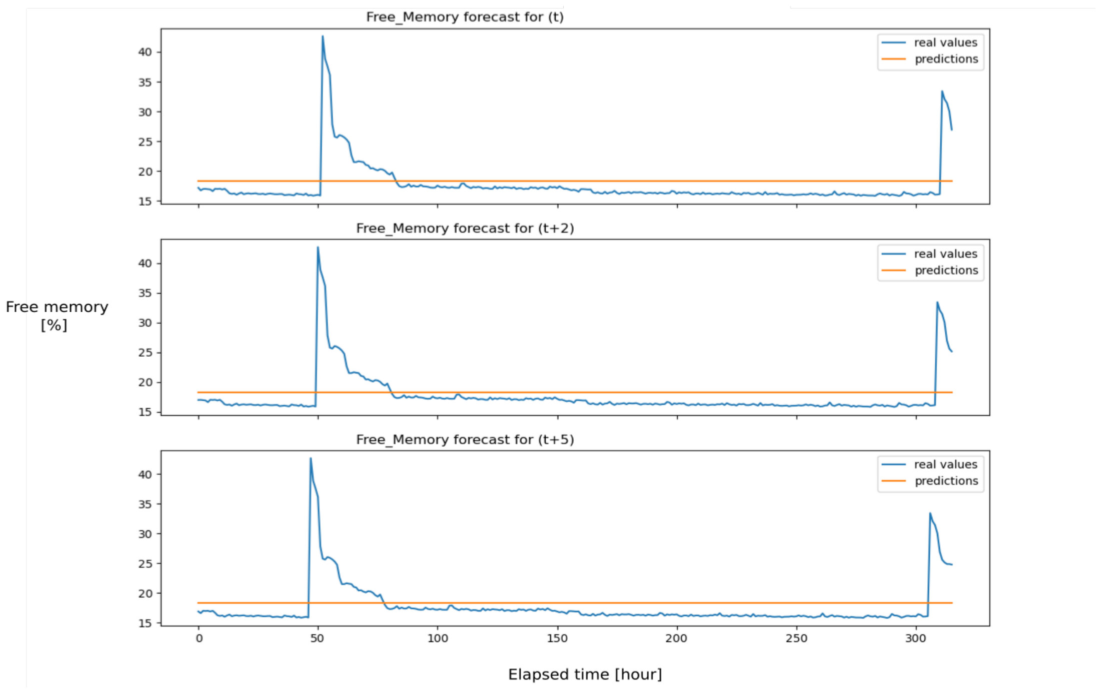


Figure 5.5: Forecasts of the vanilla LSTM model with 20 hidden nodes and 1 single layer.

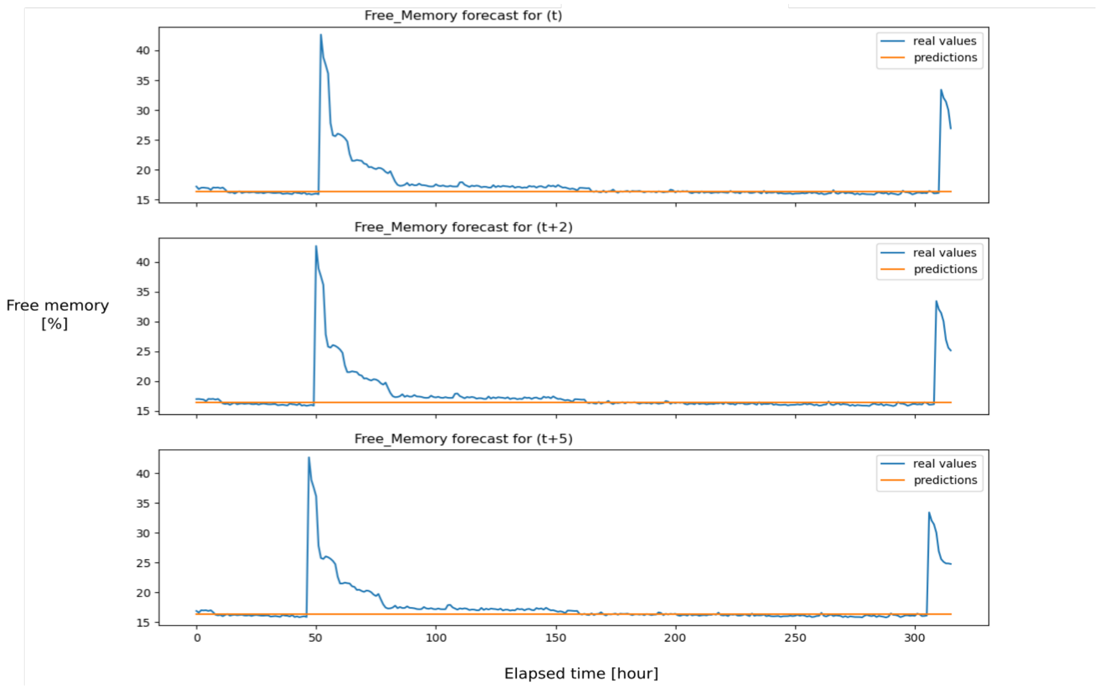
regardless of the inputs as well, even with one single layer and few nodes (20). When increasing the number of hidden nodes and the number of stacked layers, these last models show the exact same behaviour. Which probably means that the most information that can be taken out of the training data is that a constant forecast is the one that will result in a lower MSE. As a representative example of these models, in Figure 5.6(b) is a graphical representation of the forecasts computed by an encoder-decoder LSTM model with 20 hidden nodes and 1 layer.

In terms of output shapes (oscillations, naive approaches and constant outputs), the RNN cells of the three types studied all reveal similar results. This way, all of the graphical examples given for architectures with LSTM cells can be generalised for the Elman RNN and the GRU.

Despite the fact that these conclusions might be useful to understand how the models learned to compute the forecasts, they are not useful to production environments because these forecasts do not add any value beyond the information that is already visible in the data from the past.



(a) Forecasts of the vanilla LSTM model with 3 layers of 200, 500 and 50 hidden nodes, respectively.



(b) Forecasts of the Encoder-decoder LSTM model with 20 hidden nodes and 1 single layer.

Figure 5.6: Constant forecasts computed by the larger and more robust architectures.

5.1.1 Period Reduction Trial

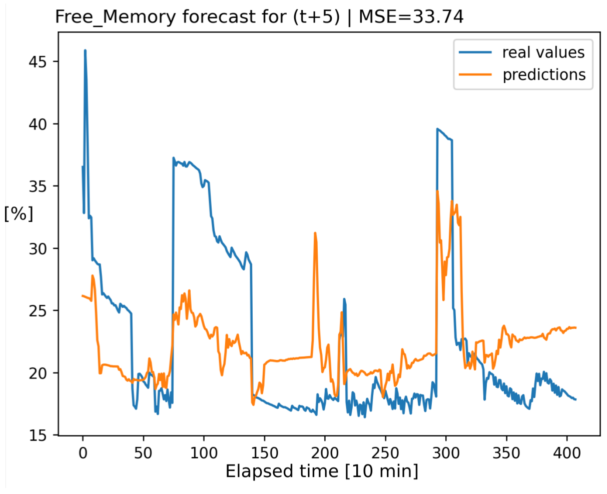
The period reduction trial experiment, described in Section 4.1.4, was also thoroughly tested with all of the ANN architectures, but the results were not promising and were very unstable. Not only it did not achieve satisfying forecasts, but also different training iterations with the same conditions – dataset, architecture and hyperparameters – led to different results, making it hard even to take conclusions from the bad results. The metrics used for this experiment were the same ones (present in Figure 5.1), but the with specifications described in Section 4.1.4.

With the heavier architectures – Encoder-Decoder, Encoder-Decoder with Attention and the Transformer – for the most part of the (repeated) experiments, the models outputted constant values, similar to the ones obtained in Figure 5.6(b). As such, either these models ignored the spikes count as an extra feature, or what they learned from the spikes simply reinforced what had been learned from the plain data – outputting a constant will likely minimize the loss function (MSE) across new datasets.

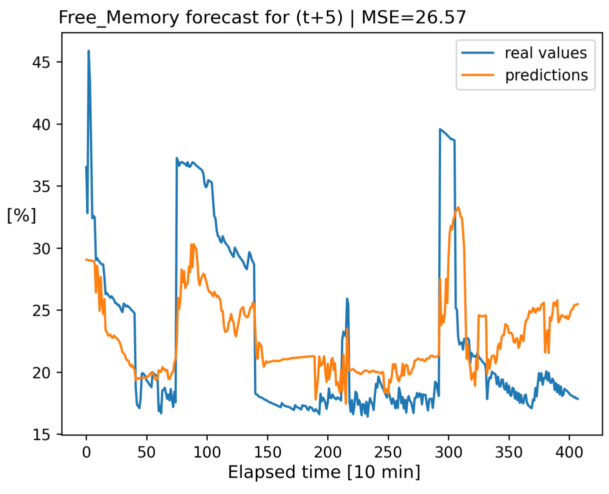
For the FNN and vanilla RNN variants, the period reduction trial revealed visible differences. Regarding the evaluation metric used (MSE), for the RNN architectures some tests led to better results than the ones in Table 5.2 and others did not, given that different training iterations constantly led to different results. For the FNN however, with 2 hidden layers of 45 and 75 hidden nodes, respectively, the results were slightly better (even considering the inconsistent repetitions) – with MSE values in between $[20, 35]$ for the last point of the output sequence (the 6th), for the free RAM memory forecast. Nonetheless, given that the time-steps of the sequences in this experiment are 10min apart (as described in Section 4.1.4), the last step predicted $(t+5)$, is only 1 hour in the future instead of 6. This way, it is hard and inaccurate to compare these values with the ones from Table 5.2. Experiments with longer sequences to match the time-horizons of the results in Table 5.2 were not performed due to the lack of available data of metrics collected with periods of 1 second. If such data was available, it is quite possible that the results would degrade significantly given that they would have to perform a multistep-ahead forecast of 36 time-steps in the future.

Moreover, it is notable that the FNN in this period reduction trial approach, reacts steeply to spikes in data and, at times, is able to detect them ahead of time, although still with high MSE values overall. Furthermore, it was evident that the threshold T defined to consider the spikes highly influences the "sensitivity" towards spikes. In Figure 5.7 is a comparison of a FNN trained with a threshold $T = 7$ – Figure 5.7(a); and the same FNN trained with $T = 15$ Figure 5.7(b). A lower threshold will count more spikes, as it will require a lower amplitude to be considered and counted as a spike. And vice-versa for a higher threshold.

From Figure 5.7(a) it is clear that the network was able to forecast steep oscillations (although not with the correct amplitude). However, it also does forecast some oscillations shortly before time (for example, around the time-step 230), and even some oscillations that did not take place at all (for example, around



(a) FNN model forecast of free RAM memory with a spike detection threshold of $T = 7$.



(b) FNN model forecast of free RAM memory with a spike detection threshold of $T = 15$.

Figure 5.7: Graphical comparison of the forecasts over different threshold values for the period reduction trial.

the time-step 190). In Figure 5.7(b), with a less sensitive threshold line, the model does not forecast steep oscillations so easily and instead tries to follow the metric's previous values. Although this second one does not seem to add any valuable forecasting information, it does result in a lower MSE – 26.57, which is significantly lower than 33.74 for Figure 5.7(a).

Despite the fact that, for the reasons explained in Section 4.1.4, this approach could not be put into production, the goal of this trial was: trying to figure out if the data from the trend tables and with large collection periods ($>>$ 1 second) could have been losing important information from the metrics continuous behaviour, and if such a loss would impede accurate forecasts from the intelligent models. The results obtained are not conclusive enough to answer those questions, but a possibility to test this hypothesis with more data should not be discarded.

5.2 Event Prediction

The work developed for the event prediction problems was also deployed in production environments and tested across several different machines as well as with different input metrics and events. To illustrate the behaviour and performance of the algorithms developed for event prediction, the set of input metrics of Figure 5.1 was used alongside two different events S_1 and S_2 that the models learned do predict. These events are related to problems in IT systems that are useful to be predicted in advance of the actual occurrence. Both S_1 and S_2 are referent to services running on the same machine where the metrics were collected, service 1 and service 2, respectively. Whenever service 1 crashes or is down, the event S_1 is triggered, and when it is back up and running, the event S_1 is signalled and reverts the trigger, producing this way a dataset of the form of Table 1.3. For service 2, the same mechanism is used with the correspondent event S_2. For the event prediction problems, the dataset split into training set and test set was also of 80% and 20%, respectively. To have a clearer visualization of the occurrence of events alongside IT systems, one can look at the example of Figure 4.7.

The ANN models introduced in Chapter 3 were also experimented for a vast range of hidden nodes and stacked up layers, following the implementation techniques described in Section 4.2. In Table 5.3 is a summary of the intervals tested for each architecture.

Table 5.3: Ranges of hyperparameters experimented for the event prediction models.

Event prediction model	Hidden nodes	Stacked up layers
FNN	[15, 100]	[1, 6]
Vanilla RNN variants	[15, 500]	[1, 3]
Encoder-decoder variants	[15, 150]	[1, 3]
Attention Encoder-decoder variants	[15, 150]	[1, 3]
Transformer	8	[1, 10]

Again, the variants are referent to the three RNN cells studied in this work – basic RNN, LSTM and GRU. The iterative tests over the number of hidden nodes were carried with steps of 5 nodes per iteration.

The best performing configurations for each model are present in Table 5.4, alongside the correspondent evaluation metrics described in Section 4.2.3.A – precision and recall.

Table 5.4: Best performing configurations for the event prediction models – with the corresponding precision and recall.

Event prediction Model	Hidden nodes	Stacked up layers	S_1		S_2	
			Precision	Recall	Precision	Recall
FNN	45 — 60	2	48.55	55.95	47.75	58.94
Vanilla RNN	35	1	36.72	35.50	33.68	34.82
Vanilla LSTM	25	1	43.84	44.16	45.07	50.95
Vanilla GRU	30	1	39.95	41.59	44.83	44.38
Encoder-decoder RNN	25	1	33.57	35.29	31.33	35.41
Encoder-decoder LSTM	20	1	36.35	34.77	35.16	35.67
Encoder-decoder GRU	15	1	34.17	34.16	36.76	32.15
Attention Encoder-decoder RNN	30	1	37.19	32.86	34.11	34.46
Attention Encoder-decoder LSTM	25	1	41.84	32.89	38.23	37.57
Attention Encoder-decoder GRU	30	1	38.11	31.12	36.09	37.30
Transformer	8	6	45.28	40.70	48.34	40.96

In order to have a better visual understanding of how the models performed over the two events S_1 and S_2, Table 5.5 shows the average performance at the two services for each architecture.

The event prediction algorithms' results are not ideal, but already provide useful outputs and show that the algorithms were able to learn from the input data on how to predict events. The FNN is the undisputed winner with the techniques developed for these problems, in both precision and recall mea-

Table 5.5: Average evaluation of the models of Table 5.4 for the events S_1 and S_2.

Model	$(S_1 + S_2)/2$	
	Precision	**Recall**
FNN	48.15	57.45
Vanilla RNN	35.20	35.16
Vanilla LSTM	44.46	47.56
Vanilla GRU	42.39	42.99
Encoder-decoder RNN	32.45	35.35
Encoder-decoder LSTM	35.75	35.22
Encoder-decoder GRU	35.47	33.16
Attention Encoder-decoder RNN	35.65	33.67
Attention Encoder-decoder LSTM	40.03	35.23
Attention Encoder-decoder GRU	37.10	34.21
Transformer	46.81	40.83

surements. The vanilla LSTM and GRU networks also performed reasonably and, surprisingly, the Transformer outperformed all of the other encoder-decoder models.

Given that the oscillation detections (described in Section 4.2) were the key for the models to learn to predict the event occurrences, the FNN managed to take better advantage of them due to the fully connected layers that compose its architecture (Section 3.2.1). The RNN variants, on the other hand, process the inputs sequentially, which attenuates the presence of the spikes over the network. The fact that the encoder-decoder architectures performed bellow the vanilla RNN only strengthens this hypothesis, given that the inputs have to go through two entire RNNs. The attention mechanism still caught up some information that faded through the encoder-decoder but still underperformed the vanilla RNN models. For last, the Transformer does not have the feedback loop of all the RNN cells based models and thus is able to better capture the oscillations information from the inputs. Nonetheless, it still underperformed the integral fully connected layers of the FNN model, that are able to interpret the oscillation detection inputs and directly forward it to predict the events.

The emphasis on the importance of the oscillation identification by the standard deviations must be highlighted. Before the implementation of this feature, all of the methods had significantly worst results. In Table 5.6 is the performance of the same studied architectures, but without this feature – the inputs are solely sliding windows of the metrics datasets.

Moreover, the minority class oversampling technique used for all the above results was implemented with a ratio of $R_{OMC} = 0.5$. This parameter highly influences the outputs and might be important to adjust it according to the programmer's preferences and needs. If the ratio is increased, the model will be fed with more event occurrences, which will result in a lower precision but a higher recall – if the model is going to "risk" more often that an event will occur it will give false alerts more often (lower precision), however, it will let fewer occurrences pass under the radar (higher recall). On the other hand, if the ratio R_{MOC} is decreased, the model will be fed with fewer event occurrences, which will result in the opposite – higher precision and lower recall – if the model learns from positive occurrences fewer times, it will be

Table 5.6: Average evaluation of the models for the events S_1 and S_2, without the standard deviation preprocessing for oscillation detection.

Model	$(S_1 + S_2)/2$	
	Precision	**Recall**
FNN	12.66	8.54
Vanilla RNN	15.75	18.50
Vanilla LSTM	21.61	22.63
Vanilla GRU	19.98	20.47
Encoder-decoder RNN	16.47	19.02
Encoder-decoder LSTM	26.11	22.13
Encoder-decoder GRU	24.27	23.62
Attention Encoder-decoder RNN	17.71	16.14
Attention Encoder-decoder LSTM	27.43	22.22
Attention Encoder-decoder GRU	22.24	24.51
Transformer	20.10	21.14

more conservative in predicting occurrences, and will only do it when has a higher degree of confidence (higher precision), but will let more event occurrences pass under the radar (lower recall). In Table 5.7 are illustrative results of experiments with difference ratio R_{MOC} values by the best performing model – FNN – for the event prediction problems.

Table 5.7: R_{OMC} variations for event predictions with the FNN model of Table 5.4.

R_{OMC}	$(S_1 + S_2)/2$	
	Precision	**Recall**
0.25	69.19	38.41
0.5	48.15	57.45
0.75	37.08	73.21

The use of different R_{OMC} values can be of great use because it allows directing the model to a deliberated bias. If, for example, a particular event is linked to a failure that can cause severe damages, one could increase the R_{OMC} ratio to avoid not being alerted at all costs; if, on the other hand, an event is not linked to severely harmful causes or if it is too demanding to protect/prevent, one could lower the R_{OMC} to only be alerted with a higher degree of certainty.

6

Conclusion

Contents

6.1 Conclusions

The algorithms developed in this dissertation were focused on real-world use cases. They were deployed in a production environment, where they acted on data prevenient from machines working in the IT industry. The main goal of the developed work was to perform PdM – make predictions about the monitored systems that could help take preventive actions to maintain them. For this purpose, a study of practical interests was operated together with this dissertation's partnership company - Identity -, two different approaches were taken:

- **Time-series forecasting** – A deep background study was conducted to extract from the literature, the most promising methods to perform this task. The most advanced techniques developed for sequence modelling (of which time-series is part of) are the NLP models like encoder-decoders and Transformers. Therefore, these methods were tested, upon the required modifications for time-series problems. However, these advanced and complex models could not outperform the vanilla RNN models, namely the LSTM which is the model that scored the lowest MSE in tests. Unfortunately, none of the models tested in this work delivered encouraging results. The reason could have either been that the algorithms were not suitable to model the data, or that the data itself is not capable of establishing causal factors. Consequently, each in their own way, the models ended up learning naive approaches or to just output a constant in order to minimize the loss function, regardless of the "reasonless" outputs. Moreover, given the disappointing results in this section, an alternative data preprocessing/rearrangement technique was employed (described in Section 4.1.4), to try to make sense of spikes present in the data and maybe justify the bad results. Moreover, this experiment was not intended to be put into production, as it was incompatible with the monitoring platform where the algorithms were deployed. However, due to the scarce data needed for this experiment, solid conclusions could not be taken. Nonetheless, the spikes counting technique helped the FNNs to forecast steep oscillations in data, that could not have been done before (even though these forecasts were unstable).

- **Event Prediction using time-series** – Given the similarities in the nature of data of this field, with time-series forecasting problems, the state-of-the-art study for event prediction intelligent methods shares most part of the algorithms with the forecasting field. So much so that, the literature shows that not only there are already studies extrapolating the forecasting science to event prediction [65], but also the evolution of the event prediction techniques over time [66] comprises more or less the same evolution seen in time-series forecasting. This way, the same ML architectures tested in the forecasting domain were employed in the event prediction study. Moreover, a visual data analysis of the problem recognized that often event occurrences were surrounded by data oscillations on the metrics collected on the same machine as the events. Accordingly, a method to detect these oscillations (with standard deviation) and explicitly feed that information to the model was developed (described in Section 4.2.2.D). The oscillations detection approach significantly improved the predictions, that before this treatment had very poor performances. As reported in Section 5.2, the predictions are not ideal, but their outputs can be useful and are already capable of being interpreted. The FNN is the model that revealed the best performances, as it can take batter advantage of the oscillation detection data thanks to the fully connected layers that composes its architecture. The recurrent networks end up attenuating the oscillation detection inputs in their feedback loops, resulting in worse results than the simpler FNN. The Transformer does not comprise any recurrent mechanisms and thus is able to forward more easily the oscillation detection inputs to the outputs of the network. Even though still behind the FNN, the Transformer outperformed the other methods based in recurrent units. Lastly, the oversampling minority class technique can be manipulated, by tuning the ratio R_{OMC} as desired, which will result in projecting a deliberate bias in the predictions, to either be riskier or more conservative (complete explanation in Section 5.2).

6.2 Future Work

To continue the work developed in this dissertation, some suggestions can be made that, if successful, would add value to this study:

- **Automation** – An automated method to find out the best sets of metrics to perform the forecasts and the event predictions and another automated approach to select the optimal sequence lengths to feed the network.
- **Forecast spikes** – If a model that perform accurate forecasts is found, tune it to forecast spikes optimally. In PdM it is much more valuable to forecast unexpected behaviours than just achieving good accuracies when everything is normal and running as expected.
- **Period reduction trial** – Gather more historical data to try to make reliable conclusions out of this experiment (described in Section 4.1.4).
- **Event Prediction after forecasting** – If a model that performs accurate forecasts is found, explore the possibility of using the forecasts to predict events.
- **Event Prediction explanation** – Develop a mechanism that identifies which metrics unexpected behaviours contribute to the prediction of an event.

Bibliography

[1] TIME SERIES ANALYSIS & FORECASTING – mathematica-city. [Online]. Available: http://mathematicacity.co.in/2020/08/time-series-analysis-forecasting-siddhartha/

[2] M. Deshp and e. (2020) Perceptrons: The first neural networks. Section: Machine Learning. [Online]. Available: https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/

[3] L. Coelho Junior, J. Rezende, A. Batista, A. Mendonça, and W. Lacerda, "Use of artificial neural networks for prognosis of charcoal prices in minas gerais," 2013, journal Abbreviation: CERNE Publication Title: CERNE Volume: 19.

[4] "Recurrent neural network," page Version ID: 1000541866. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Recurrent_neural_network&oldid=1000541866

[5] 9.6. encoder-decoder architecture — dive into deep learning 0.16.1 documentation. [Online]. Available: https://d2l.ai/chapter_recurrent-modern/encoder-decoder.html

[6] S. Kostadinov. (2019) Understanding encoder-decoder sequence to sequence model. [Online]. Available: https://towardsdatascience.com/understanding-encoder-decoder-sequence-to-sequence-model-679e04af4346

[7] (2019) Differentiable programming and its applications to dynamical systems. [Online]. Available: https://www.groundai.com/project/differentiable-programming-and-its-applications-to-dynamical-systems/1

[8] (2019) Attention mechanism. [Online]. Available: https://blog.floydhub.com/attention-mechanism/

[9] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[10] B. Marr. (2018) How much data do we create every day? the mind-blowing stats everyone should read. Section: Enterprise & Cloud. [Online]. Available: https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/

[11] (2019) Importance and benefits of predictive and preventive maintenance. [Online]. Available: https://www.eaglecmms.com/importance-and-benefits-of-predictive-and-preventive-maintenance/

[12] "Time series," 2020, page Version ID: 993102431. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Time_series&oldid=993102431

[13] (2018) State of deep learning: Industrial PdM | LinkedIn. [Online]. Available: https://www.linkedin.com/pulse/deep-learning-iiot-checklist-lothar-schubert/

[14] R. Adhikari and R. K. Agrawal, "An introductory study on time series modeling and forecasting," 2013. [Online]. Available: http://arxiv.org/abs/1302.6613

[15] G. Weiss and H. Hirsh, "Learning to predict rare events in event sequences," 1998.

[16] G. Bontempi, S. Ben Taieb, and Y.-A. Le Borgne, "Machine learning strategies for time series forecasting," in *Business Intelligence*, M.-A. Aufaure and E. Zimányi, Eds. Springer Berlin Heidelberg, 2013, vol. 138, pp. 62–77, series Title: Lecture Notes in Business Information Processing. [Online]. Available: http://link.springer.com/10.1007/978-3-642-36318-4_3

[17] J. G. De Gooijer and R. J. Hyndman, "25 years of time series forecasting," vol. 22, no. 3, pp. 443–473, 2006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169207006000021

[18] M. Bose and K. Mali, "Designing fuzzy time series forecasting models: A survey," vol. 111, pp. 78–99, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0888613X18306376

[19] H. Ismail Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," vol. 33, no. 4, pp. 917–963, 2019. [Online]. Available: https://doi.org/10.1007/s10618-019-00619-1

[20] V. Ravi, D. Pradeepkumar, and K. Deb, "Financial time series prediction using hybrids of chaos theory, multi-layer perceptron and multi-objective evolutionary algorithms," vol. 36, pp. 136–149, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210650217300822

[21] M. Stepnicka, J. Peralta Donate, P. Cortez, L. Vavrikova, and G. Gutierrez, "Forecasting seasonal time series with computational intelligence: contribution of a combination of distinct methods." in *Proceedings of the 7th conference of the European Society for Fuzzy Logic and Technology (EUSFLAT-2011)*. Atlantis Press, 2011. [Online]. Available: http://www.atlantis-press.com/php/paper-details.php?id=2192

[22] R. Majid and S. Mir, “Advances in statistical forecasting methods: An overview,” vol. 63, pp. 815–831, 2018.

[23] K. Bandara, C. Bergmeir, and S. Smyl, “Forecasting across time series databases using recurrent neural networks on groups of similar series: A clustering approach,” 2018. [Online]. Available: http://arxiv.org/abs/1710.03222

[24] Y. Zhang, C. Cheng, R. Cao, G. Li, J. Shen, and X. Wu, “Multivariate probabilistic forecasting and its performance’s impacts on long-term dispatch of hydro-wind hybrid systems,” p. 116243, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306261920316378

[25] F. Ma, M. I. M. Wahab, D. Huang, and W. Xu, “Forecasting the realized volatility of the oil futures market: A regime switching approach,” vol. 67, pp. 136–145, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140988317302608

[26] L. Poulos, A. Kvanli, and R. Pavur, “A comparison of the accuracy of the box-jenkins method with that of automated forecasting methods,” vol. 3, no. 2, pp. 261–267, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0169207087900070

[27] P. A. Texter and J. K. Ord, “Forecasting using automatic identification procedures: A comparative analysis,” vol. 5, no. 2, pp. 209–215, 1989. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0169207089900885

[28] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, “Statistical and machine learning forecasting methods: Concerns and ways forward,” vol. 13, no. 3, 2018.

[29] 2020, title = Comparison of statistical and machine learning methods for daily SKU demand forecasting, doi = 10.1007/s12351-020-00605-2, abstract = Daily SKU demand forecasting is a challenging task as it usually involves predicting irregular series that are characterized by intermittency and erraticness. This is particularly true when forecasting at low cross-sectional levels, such as at a store or warehouse level, or dealing with slow-moving items. Yet, accurate forecasts are necessary for supporting inventory holding and replenishment decisions. This task is typically addressed by utilizing well-established statistical methods, such as the Croston’s method and its variants. More recently, Machine Learning (ML) methods have been proposed as an alternative to statistical ones, but their superiority remains under question. This paper sheds some light in that direction by comparing the forecasting performance of various ML methods, trained both in a series-by-series and a cross-learning fashion, to that of statistical methods using a large set of real daily SKU demand data. Our results indicate that some ML methods do provide better forecasts, both in terms of accuracy and bias. Cross-learning across multiple SKUs has also proven to be beneficial for some of the ML methods. © 2020, Springer-Verlag GmbH Germany, part

of Springer Nature., journaltitle = Operational Research, author = Spiliotis, E. and Makridakis, S. and Semenoglou, A.-A. and Assimakopoulos, V., date = 2020, keywords = Cross-learning, Forecasting accuracy, Neural networks, Regression trees, SKU demand, file = SCOPUS Snapshot:/home/pedromoreira/Zotero/storage/5Q45BY33/display.html:text/html,.

[30] N. K. Ahmed, A. F. Atiya, N. E. Gayar, and H. El-Shishiny, "An empirical comparison of machine learning models for time series forecasting," vol. 29, no. 5, pp. 594–621, 2010. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/07474938.2010.481556

[31] G. Bontempi, S. Ben Taieb, and Y.-A. Le Borgne, "Machine learning strategies for time series forecasting," in *Lecture Notes in Business Information Processing*, 2013, vol. 138, journal Abbreviation: Lecture Notes in Business Information Processing.

[32] S. S. Kathait, D. A. Kaur, S. Tiwari, and A. Varshney, "Integrating neural networks with time series forecasting: Improving sales," vol. 04, no. 1, p. 3, 2020. [Online]. Available: https://valiancesolutions.com/whitepapers/integrating-sales-forecastingtime-series-using-neural-networks/

[33] R. Nagarajan, "Deciphering dynamical nonlinearities in short time series using recurrent neural networks," vol. 9, no. 1, p. 14158, 2019. [Online]. Available: http://www.nature.com/articles/s41598-019-50625-y

[34] D. Niu, H. Shi, J. Li, and Y. Wei, "Research on short-term power load time series forecasting model based on BP neural network," in *2010 2nd International Conference on Advanced Computer Control*, vol. 4, 2010, pp. 509–512.

[35] M. Ghofrani, D. Carson, and M. Ghayekhloo, "Hybrid clustering-time series-bayesian neural network short-term load forecasting method," in *2016 North American Power Symposium (NAPS)*, 2016, pp. 1–5.

[36] Q. Huang, Y. Li, S. Liu, and P. Liu, "Hourly load forecasting model based on real-time meteorological analysis," in *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN)*, 2016, pp. 488–492, ISSN: 2472-7555.

[37] X. Guo, X. Liang, and X. Li, "A stock pattern recognition algorithm based on neural networks," in *Third International Conference on Natural Computation (ICNC 2007)*, vol. 2, 2007, pp. 518–522, ISSN: 2157-9563.

[38] R. M. K. T. Rathnayaka, D. M. K. N. Seneviratna, W. Jianguo, and H. I. Arumawadu, "A hybrid statistical approach for stock market forecasting based on artificial neural network and ARIMA time series models," in *2015 International Conference on Behavioral, Economic and Socio-cultural Computing (BESC)*, 2015, pp. 54–60.

[39] N. S. Khan, S. Ghani, and S. Haider, “Real-time analysis of a sensor’s data for automated decision making in an IoT-based smart home,” vol. 18, no. 6, 2018. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6022067/

[40] P. Zhang, E. Patuwo, and M. Hu, “Forecasting with artificial neural networks: The state of the art,” vol. 14, pp. 35–62, 1998.

[41] G. P. Zhang, B. E. Patuwo, and M. Y. Hu, “A simulation study of arti”cial neural networks for nonlinear time-series forecasting,” p. 16, 2001.

[42] Z. Tang, C. de Almeida, and P. A. Fishwick, “Time series forecasting using neural networks vs. box- jenkins methodology,” vol. 57, no. 5, pp. 303–310, 1991, publisher: SAGE Publications Ltd STM. [Online]. Available: https://doi.org/10.1177/003754979105700508

[43] P. Zhang and D. Kline, “Quarterly time-series forecasting with neural networks,” vol. 18, pp. 1800–1814, 2007.

[44] P. Mandal, T. Senjyu, N. Urasaki, and T. Funabashi, “A neural network based several-hour-ahead electric load forecasting using similar days approach,” vol. 28, pp. 367–373, 2006.

[45] P. Zhang, “Zhang, g.p.: Time series forecasting using a hybrid ARIMA and neural network model. neurocomputing 50, 159-175,” vol. 50, pp. 159–175, 2003.

[46] A. Schäfer and H. Zimmermann, *Recurrent neural networks are universal approximators*, 2006, pages: 640.

[47] J. L. Elman, “Finding structure in time,” vol. 14, no. 2, pp. 179–211, 1990. [Online]. Available: http://www.sciencedirect.com/science/article/pii/036402139090002E

[48] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” vol. 9, pp. 1735–80, 1997.

[49] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using RNN encoder-decoder for statistical machine translation,” 2014. [Online]. Available: http://arxiv.org/abs/1406.1078

[50] R. Jozefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” p. 9, 2015.

[51] F. M. Bianchi, E. Maiorino, M. C. Kampffmeyer, A. Rizzi, and R. Jenssen, “An overview and comparative analysis of recurrent neural networks for short term load forecasting,” 2018. [Online]. Available: http://arxiv.org/abs/1705.04378

[52] M. Schuster and K. Paliwal, “Bidirectional recurrent neural networks,” vol. 45, pp. 2673–2681, 1997.

[53] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” 2014. [Online]. Available: http://arxiv.org/abs/1409.3215

[54] Y. Zhu, W. Zhang, Y. Chen, and H. Gao, “A novel approach to workload prediction using attention-based LSTM encoder-decoder network in cloud environment,” vol. 2019, no. 1, p. 274, 2019. [Online]. Available: https://doi.org/10.1186/s13638-019-1605-z

[55] T. Luong, H. Pham, and C. D. Manning, “Effective approaches to attention-based neural machine translation,” in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2015, pp. 1412–1421. [Online]. Available: https://www.aclweb.org/anthology/D15-1166

[56] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” 2016. [Online]. Available: http://arxiv.org/abs/1409.0473

[57] Y. Qin, D. Song, H. Chen, W. Cheng, G. Jiang, and G. Cottrell, “A dual-stage attention-based recurrent neural network for time series prediction,” 2017. [Online]. Available: http://arxiv.org/abs/1704.02971

[58] A. F. T. Martins and R. F. Astudillo, “From softmax to sparsemax: A sparse model of attention and multi-label classification,” 2016. [Online]. Available: http://arxiv.org/abs/1602.02068

[59] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, N. Shazeer, I. Simon, C. Hawthorne, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, “Music transformer,” 2018. [Online]. Available: http://arxiv.org/abs/1809.04281

[60] N. Parmar, A. Vaswani, J. Uszkoreit, Ł. Kaiser, N. Shazeer, A. Ku, and D. Tran, “Image transformer,” 2018. [Online]. Available: http://arxiv.org/abs/1802.05751

[61] M. Di Gangi, M. Negri, R. Cattoni, R. Dessi, and M. Turchi, *Enhancing Transformer for End-to-end Speech-to-Text Translation*, 2019.

[62] M. R. Saradjian and M. Akhoondzadeh, “Thermal anomalies detection before strong earthquakes (m $>$ 6.0) using interquartile, wavelet and kalman filter methods,” vol. 11, pp. 1099–1108, 2011.

[63] Y. Zhang, R. Xiong, H. He, and M. G. Pecht, “Long short-term memory recurrent neural network for remaining useful life prediction of lithium-ion batteries,” *IEEE Transactions on Vehicular Technology*, vol. 67, no. 7, pp. 5695–5705, 2018.

[64] P. Malhotra, L. Vig, G. Shroff, and P. Agarwal, “Long short term memory networks for anomaly detection in time series,” p. 6, 2015.

[65] N. A. Boukary, “A comparison of time series forecasting learning algorithms on the task of predicting event timing une comparaison des algorithms,” p. 97, 2016.

[66] L. Zhao, “Event prediction in the big data era: A systematic survey,” 2020. [Online]. Available: http://arxiv.org/abs/2007.09815

[67] S. Shin and H.-j. Kim, *Autoencoder-based One-class Classification Technique for Event Prediction*, 2019, journal Abbreviation: CCIOT 2019: Proceedings of the 2019 4th International Conference on Cloud Computing and Internet of Things Pages: 58 Publication Title: CCIOT 2019: Proceedings of the 2019 4th International Conference on Cloud Computing and Internet of Things.

[68] G. Hughes, J. Murray, K. Kreutz-Delgado, and C. Elkan, “Improved disk-drive failure warnings,” vol. 51, pp. 350–357, 2002.

[69] M. Shao, J. Li, F. Chen, H. Huang, S. Zhang, and X. Chen, *An Efficient Approach to Event Detection and Forecasting in Dynamic Multivariate Social Media Networks*, 2017, pages: 1639.

[70] B. W. Yap, K. A. Rani, H. A. A. Rahman, S. Fong, Z. Khairudin, and N. N. Abdullah, “An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets,” in *Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013)*, ser. Lecture Notes in Electrical Engineering, T. Herawan, M. M. Deris, and J. Abawajy, Eds. Springer, 2014, pp. 13–22.

[71] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[72] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” 2016. [Online]. Available: http://arxiv.org/abs/1607.06450

[73] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional sequence to sequence learning,” 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[74] D. Britz, A. Goldie, M.-T. Luong, and Q. Le, “Massive exploration of neural machine translation architectures,” 2017. [Online]. Available: http://arxiv.org/abs/1703.03906

[75] J. Snoek, H. Larochelle, and R. Adams, “Practical bayesian optimization of machine learning algorithms,” vol. 4, 2012.

[76] J. Bergstra and Y. Bengio, “Random search for hyper-parameter optimization,” p. 25, 2012.

[77] N. Qian, “On the momentum term in gradient descent learning algorithms,” vol. 12, no. 1, pp. 145–151, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608098001166

[78] Neural networks and deep learning. [Online]. Available: https://www.coursera.org/learn/neural-networks-deep-learning

[79] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017. [Online]. Available: http://arxiv.org/abs/1412.6980