Heriot-Watt University

Masters Thesis

Electricity Load Forecasting

Author:
Jonathan Meyer
Student number:
H00281038

Supervisor:
Dr. Katrin Solveig Lohan
Second reader:
Dr. Hamish Taylor

A thesis submitted in fulfillment of the requirements for the degree of MSc. in Artificial Intelligence with SMI

in the

School of Mathematical and Computer Sciences

August 2018

Declaration of Authorship

I, Jonathan MEYER, declare that this thesis titled, 'Electricity Load Forecasting' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed: 

Date: 13/08/2018

Abstract

Electricity is the essence of modern life and any improvements in its delivery and usage can greatly impact people's lives. Generating electricity is no easy task, not only due to its sheer quantity but also due to the fluctuating usage throughout the day and months.

In this project, we aim at developing an electric forecasting module that will produce reliable load / demand predictions, enabling energy providers to operate at the highest reliability, lowest cost and reduced environmental footprint.

This problem was traditionally addressed using statistical methods including Autoregressive Integrated Moving Average (ARIMA) and Holt-Winters but the focus is now mostly on Computational Intelligence techniques due to their superior performance in most cases. These techniques include Machine Learning models such as Artificial Neural Networks (ANN) and more recently a number of Deep Learning models such as Recurrent Neural Networks(RNN) and Convolutional Neural Networks (CNN) exhibiting state of the art performance in many tasks.

In order to gain knowledge in the field and identify the most suitable methods, a throughout literature review will be conducted, with an emphasis on Deep Learning techniques. Then, a selection of models will be implemented and evaluated using the Keras Python API and data from National Grid UK and NYISO.

Acknowledgements

I would like to thank my project supervisor, Dr. Katrin Solveig LOHAN, for her continuous support and useful remarks. The door to Dr. LOHAN office was always open whenever I needed feedback about my research and she consistently allowed this project to be my own while steering me in the right direction.

I would also like to thank Ph.D. student SHEPPARD Eli, currently conducting his thesis at Heriot Watt University on a topic involving Machine Learning, for the time he dedicated to me and the useful advice he provided throughout this project.

Finally, I also acknowledge Dr. TAYLOR Hamish of Heriot Watt University as the second reader of this thesis and appreciate his valuable comments.

Contents

List of Figures

Abbreviations

STFL	**S**hort **T**erm **L**oad **F**orecasting
AI	**A**rtificial **I**ntelligence
ML	**M**achine **L**earning
DL	**D**eep **L**earning

Statistical models

ARIMA	**A**utoregressive **I**ntegrated **M**oving **A**verage
DSHW	**D**ouble **S**easonal **H**olt-**W**inters

Machine learning models

SVR	**S**upport **V**ector **R**egression
SVM	**S**upport **V**ector **M**achine
RF	**R**andom **F**orest

ANN	**A**rticial **N**eural **N**etwork
MLP	**M**ulti-**L**ayer **P**erceptron
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort-**T**erm **M**emory
DBN	**D**eep **B**elief **N**etwork
RBM	**R**estricted **B**oltzmann **M**achine

BP	**B**ack-**P**ropagation
SGD	**S**tochastic **G**radient **D**escent
ReLU	**Re**ctified **L**inear **U**nit

Evaluation metrics

MAE	**M**ean **A**bsolute **E**rror
MSE	**M**ean **S**quare **E**rror
RMSE	**R**oot **M**ean **S**quare **E**rror
MAPE	**M**ean **A**bsolute **P**ercentage **E**rror

Chapter 1

Introduction

The electric power system is one of the key pieces of our modern lifestyle and it is impossible for most individuals to envision a future without it. However, sustaining a stable grid is no small feat and encompasses many disciplines. Indeed, electrical grids can be regarded as the largest human-made structures, spanning across entire countries and containing thousands of components.

The involvement of Computer Science in this field is large and related to various critical aspects such as load monitoring and infrastructure control.

Indeed, one of the difficulties in this mission of providing a safe, affordable and widely available energy is the task of forecasting its demand, hence the load on the generation systems. Load forecasting is a crucial aspect of electric power generation because it enables utility providers to plan ahead the resources that are required to meet the demand. Failure to do so can lead to unnecessary expenses and power outages.

In order to understand the importance of load forecasting in the electric power industry, one has to imagine the effect of utilizing a system outside its ideal range.

While operating under specifications shouldn't lead to any catastrophic event, this over-commitment of resources increases operational costs.

On the other end, if the load is greater than the forecast, the use of expensive peaking units (such as coal-fired plants) is required. A noteworthy exception is Australia's recent Hornsdale power reserve, made of Lithium batteries.

The electrical grid operations can also be threatened by line balancing issues, as an extreme case, we can refer to the 2006 European Blackout caused by a major grid tripping.

Note that this project will be completed as part of an industrial partnership with the company Smarter Grid Solutions Ltd.

1.1 Aims and objectives

1.1.1 Aims

This dissertation aims at exploring the applicability of Deep Learning techniques to the field of electric load forecasting through theoretical and practical points of view.
The results of my investigation will be reported in the final submission of my thesis using methods described in the "Evaluation" Section of this report. Multiple models will be implemented and tested on three datasets introduced in Section 3 "Data Sources". One of the sources (NYISO) also provide load forecast, to which the developed model will be compared to.

1.1.2 Objectives

The first objective of this dissertation is to pursue a literature review regarding modern techniques applied to load forecasting in order to gain knowledge in the field. The research effort will be focused on Artificial Intelligence (AI) based techniques, especially those belonging to the Deep Learning (DL) subfield.

The second objective is to plan the development of Machine Learning (ML) models used to forecast electric load with the use of the Keras toolkit and the Python programming language.

Thirdly, a requirements analysis will be conducted, highlighting the normative requirements along with any noteworthy information with regards to the software software, hardware and personnel involved. Similarly, a risk evaluation will be carried out in order to identify and prevent, whenever possible, situations that could negatively impact the project. Professional, Legal, Ethical and Social issues will also be covered in this dissertation.

The final goal of this project is to propose a forecasting module to be integrated into Smarter Grid Solutions Ltd. Distributed Energy Resource Management System (DERMS). As such, this dissertation also has the goal of instructing Smarter Grid Solutions Ltd. about the best practices in the field of electric forecasting and, therefore, will contain a significant proportion of high-level technical details. Three datasets will be used for the training of the developed forecasting module and used as reference in order to assess the performance of the developed models.

1.2 Smarter Grid Solutions

Smarter Grid Solutions Ltd. is a Scottish company supplying electricity providers with software tools to manage various aspects of their installations. Its offer is specifically tailored for systems containing Distributed Energy Resources (DER) such as smart-grids. However, they are willing to enhance their proposition with forecasting capabilities, which is why this partnership takes place.

Chapter 2

Problem description

2.1 Electric load forecasting

Electric load forecasting involves the task of predicting the electrical power demand (in Watt - W) at a given moment in time and optionally, in space. Depending on the system setup, the data used and the main objective of said forecasting, this can be achieved using different techniques.

Usually, the aim is two-fold, first, to reduce the operational costs of the provider and second, offer a reliable energy output.

To illustrate the importance of load forecasting, Hong [2015] estimated that a 1% improvement in short-term load forecasting (STLF) error for a 1 GW power plant can lead to 300000$ savings. For reference, UK's average electric demand is upwards of 30GW.

This problem is therefore seriously studied by large electric providers such as National Grid, recently in talks with the company DeepMind known for several state of the art deep learning technologies (Alphago, Go game AI : Wang et al. [2017], Language generation, Wavenet : Silver et al. [2017]). The UK government hopes to see a 10% reduction in energy cost from this optimization process alone (Financial-Times [2017]).

For reference, on Figure 2.1, we can see a plot of a forecasted load time series using various models which is the main goal of this dissertation.

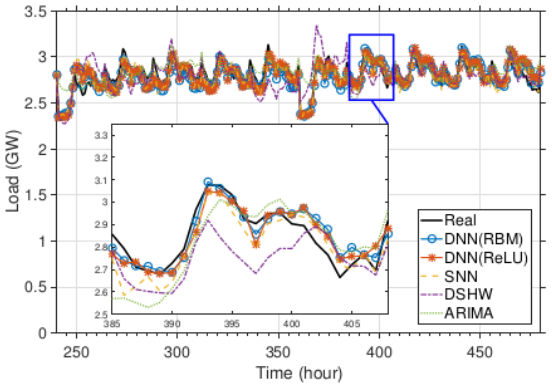


FIGURE 2.1: Load forecasting (Ryu et al. [2016])

2.1.1 Forecasting horizon

The forecasting of a time series variable (such as the electric load)" can be done on different time scales. While there is no consensus on what strictly determines these various horizons, we will employ the most common ones :

- Short-term : Refers to the forecasting of data from a few hours ahead to a week.
- Mid-term : Involves a forecasting range between a month and a year.
- Long-term : Focus on modeling from a year to several years.

Additional horizons are sometime defined in specific scenarios such as very short-term forecasting ranging from a few minutes to an hour and very long-term that can span to several decades.

The horizon is determined by the business objectives as these forecasts can be leveraged to improve various aspects of the energy industry. Short-term forecasting is mostly used for daily operations and enables providers to schedule the generation units as well as plan eventual energy transactions (with neighbouring countries for example).

On the other end, mid-term and long-term scheduling are useful for defining investments in additional infrastructures and large maintenance operations.

Unfortunately, an effective model on a given horizon is unlikely to perform best on a vastly different horizon.

Moreover, depending on the horizon, different additional variables may be considered, along with the historical load data. For example, since the weather has a strong impact on load, it is relevant to include the local temperature as a short-term model input, while this is not relevant for long-term forecasting and longer-lived variables such as subscriber base or country GDP may be of higher interest.

In this work, we will focus on Short-Term Load Forecasting (STLF), mainly day-ahead.

2.1.2 Time series

Time series can be found everywhere, from finance and astronomy to engineering. While they share common characteristics, we will focus on time series related to the energy forecasting domain. The time series of interest are :

- Load versus time : Load in MW as a function of time.
- Various other input variables such as weather information.

It should be noted that this project solely focuses on the forecasting of electricity load as such, any additional time series forecast data will come from external sources.

These time series are highly non-linear, stochastic and contain high levels or noise (variability and volatility); requiring complex models in order to produce accurate forecasts. They also contain multiple cyclical patterns, the main one being daily, weekly and seasonal cycles.

It is also important to note that time series are composite entities and can be decomposed into sub-components, namely :

- Level : Average value of the series.
- Trend component : General evolution of the series (e.g increasing, decreasing).
- Seasonal component : Reflects seasonality (e.g week, yearly seasons etc).
- Noise : Irregular, random fluctuations (remainder after other component removal).

On Figure 2.2, we can see such decomposition for a passenger time series.

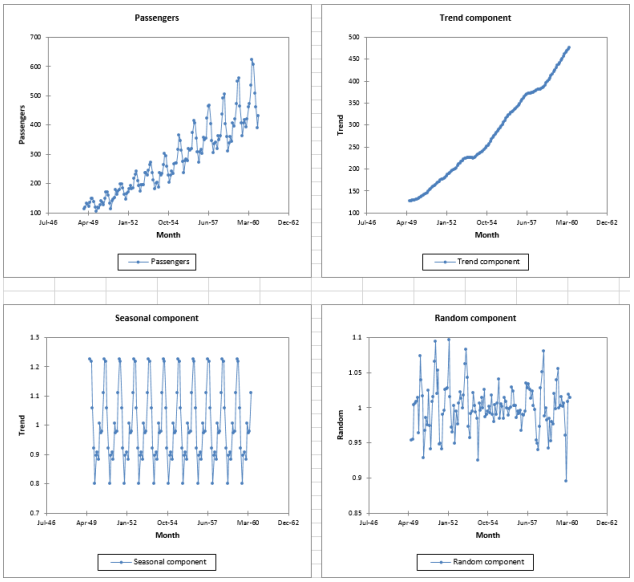


FIGURE 2.2: Time series decomposition (XLSTAT)

2.1.3 Impacts of data scale

The load on a micro-grid setup (hence, on a smart-grid too) can be magnitudes lower when compared to aggregated load datasets which usually represent a large region or even a whole country.

As pointed by Hernández et al. [2014], the load on micro-grids (hence, on smart-grids) also have sharper load peaks and troughs, making forecasting more difficult (see Figure 2.3). We can see that the load profile of a whole country is significantly different to the loads from smaller regions, especially when compared to home consumption for which the load changes very rapidly throughout the day.

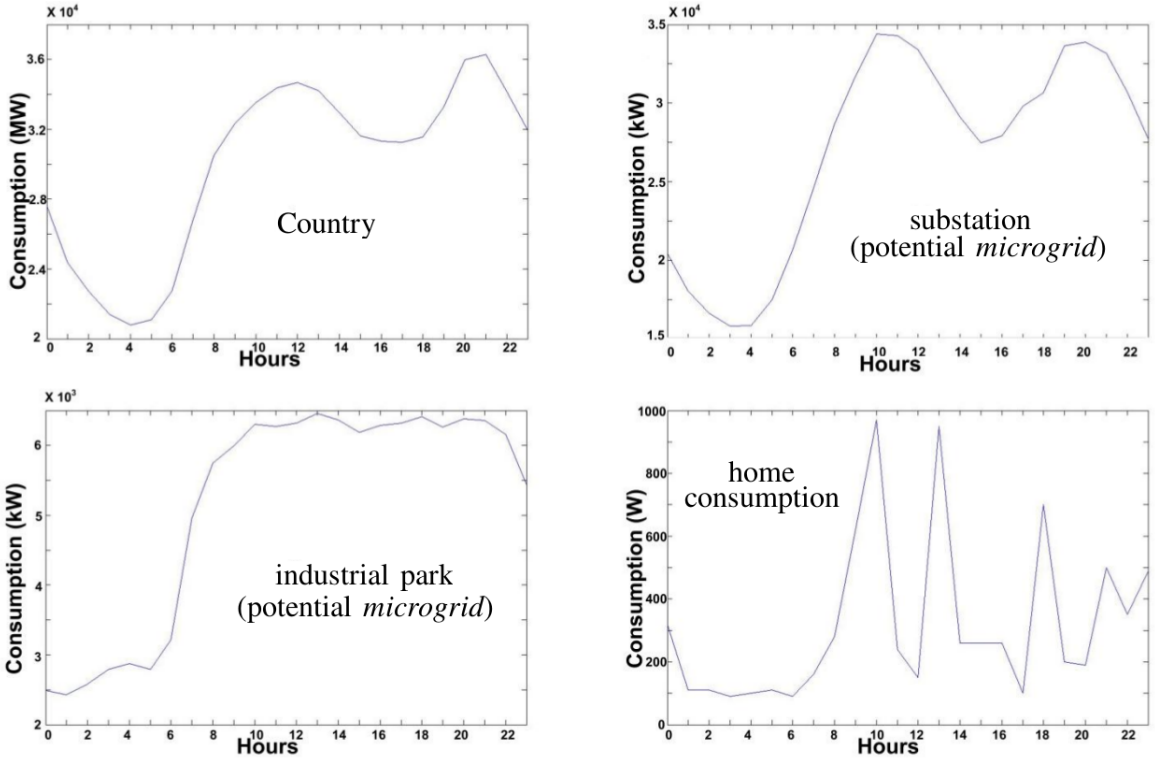


FIGURE 2.3: Load profile by aggregation level. (Hernández et al. [2014])

The faster changing time series are therefore more complex to forecast accurately. Note that one of the three datasets explored in this dissertation has a very low load, hence similar load profile to that of a micro-grid.

Chapter 3

Data sources description

Machine learning models require data to be trained on. As such, one has to find the most suitable dataset for a given task.

In this project, 2 types of datasets are used : Electricity and weather data.

Note that the electricity datasets can either refer to the load or the demand. While the "load" is characterized by the energy flow from the generator(s), the demand corresponds to the amount of energy that is effectively consumed. The load and demand can differ for 2 reasons : some consumers also produce some energy and / or some losses occur during the energy transfer between the producers and consumers. In this project, this distinction has no consequences.

3.1 Electricity datasets

Load and demand data is given in MW (megawatt). Both datasets were selected by the industrial partner. Theses 2 sources are National Grid UK, providing historical demand data for the UK, and NYISO providing load for the New York state (USA).

3.1.1 National Grid UK

The experimentation dataset is constructed from data available at NationalGrid.

More specifically, this dataset contains the historical UK national demand from April, 1^{st} 2005 to March, 16^{th} 2018 (date of data collection).

The dataset frequency is 30 minutes.

The main fields of this dataset are the following :

- "SETTLEMENT_DATE" : Date with format DD-MMM-YYYY[1,2]
- "SETTLEMENT_PERIOD" : Time of the day as an integer from 1 to 48.
- "ND" : National electricity demand.
- "ENGLAND_WALES_DEMAND" : England and Wales only electricity demand.

The complete list of fields along with their official descriptions is available at : National Grid UK data fields description.

3.1.2 NYISO

The experimentation dataset is constructed from data available at NYISO Load data.

More specifically, this dataset contains the historical load from 11 zones in the New York state (USA) from May, 26^{st} 2001 to May, 26^{th} 2018 (date of data collection).

The dataset frequency is 1 hour. Note that 5 min data is also available.

The choice of using the 1 hour based dataset ("Integrated load") was motivated by Marino et al. [2016] experiments using 1 minute and 1 hour based datasets showing lower performance when using the higher frequency dataset. Additionally, higher frequency data leads to larger dataset files, in turn increasing computational needs (RAM, processing power, storage).

NYISO zones are visible on Figure 3.1.

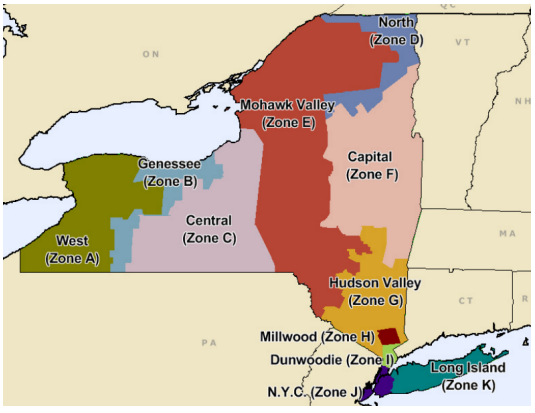


FIGURE 3.1: NYISO zone map (Federal-Energy-Regulatory-Commission [2018])

[1] e.g. 01-JAN-2018

[2] The file named "DemandData_2011-2016.csv" has a different format : DD-Mmm-YYYY (e.g. 01-Jan-2018)

The fields of this dataset are the following :

- "Time Stamp" : Date with format MM/DD/YYYY (e.g. 12/31/2018).
- "Name" : Zone name (listed in Table 3.1).
- "Integrated Load" : Electricity load.

3.1.3 Electricity datasets summary

							Detected errors			
Data source	Dataset	Start date	End date	Frequency	Data points	Average value	Zeros	Extremes	Step	Total errors
NationalGrid UK	National	01/04/2005	16/03/2018	30 minutes	225984	35144	0	0	0	0
	England +Wales	01/04/2005	16/03/2018	30 minutes	225984	31867	8	1	3	12
NYISO	CAPITL	21/06/2001	15/05/2018	1 hour	148078	1330	8	1	1	10
	DUNWOD	21/06/2001	15/05/2018	1 hour	148078	700	8	34	10	52
	HUD VL	21/06/2001	15/05/2018	1 hour	148078	1172	8	3	3	14
	MHK VL	21/06/2001	15/05/2018	1 hour	148078	880	8	8	5	21
	NORTH	21/06/2001	15/05/2018	1 hour	148078	647	8	4	14	26
	N.Y.C._LONGIL	21/06/2001	30/01/2005	1 hour	31638	8375	0	6	0	6
	CENTRL	21/06/2001	15/05/2018	1 hour	148078	1878	8	0	0	8
	GENESE	21/06/2001	15/05/2018	1 hour	148078	1137	8	0	1	9
	LONGIL	31/01/2005	15/05/2018	1 hour	116440	2519	8	27	0	35
	MILLWD	21/06/2001	15/05/2018	1 hour	148078	307	8	53	12	73
	N.Y.C	31/01/2005	15/05/2018	1 hour	116440	6111	8	0	0	8
	WEST	21/06/2001	15/05/2018	1 hour	148078	1812	8	0	0	8

TABLE 3.1: Load datasets information

Note that the "N.Y.C._LONGIL" was split on 30/01/2005 into two separate datasets named "N.Y.C" and "LONGIL".

3.1.4 Electricity datasets analysis

Of all these datasets, three were selected :

- National Grid's UK national demand : High demand, large area, large population (∼35000 MW, pop. 65M)
- NYSIO's NYC load : Medium load, small area (∼6000MW, pop. 8.5M)
- NYISO's MHK VL load : Low load, medium area (∼2000MW)

These were chosen because of their different load levels, over various surface areas.

Load / Demand per month

The following two figures seem to highlight the effect of weather on the load. Indeed, on Figure 3.2, the leftmost sub-figure shows that the UK demand is measurably lower during the summer months (gray boxes) compared to the winter months. This is opposite to the NYC dataset where the load significantly increases during the summer. Considering that the UK experiences an average high of 20°C during the summer compared to 30°C for New York, this difference in load profile could be attributed to air conditioning systems, known for their high energy consumption. When it comes to winter months, despite having similar temperatures in both locations, the UK dataset shows a proportionally larger electricity demand.

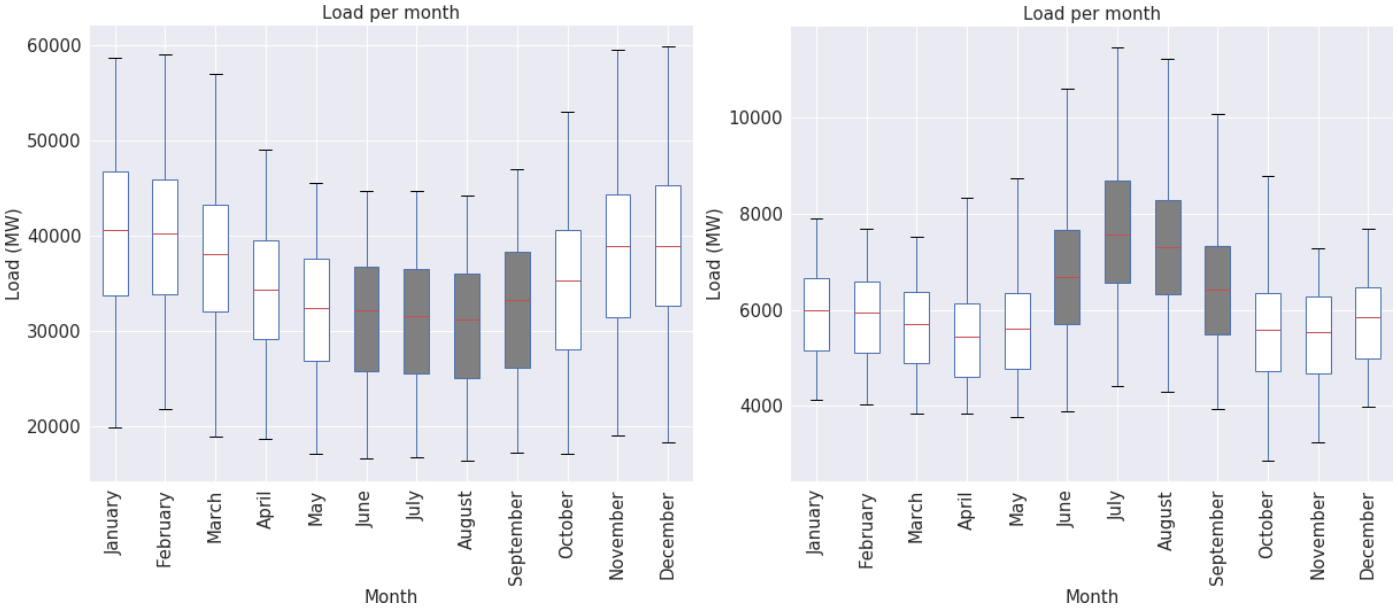


FIGURE 3.2: Monthly load / demand for UK and NYC datasets

Rate of change per month

Demand / load at smaller scales tend to be more volatile, hence more difficult to forecast.

When comparing NYC and MHK datasets (Figure 3.3, the main observation is that, while the rate of change profile is similar for both datasets (i.e. larger changes during the morning and end of afternoon), the NHK dataset exhibits larger rate of changes throughout the day with a 75% percentile peaking at a 30% load change between two timesteps (1 hour). These larger rate of change are due to the more limited number of consumers, having similar and very overlapping consumption patterns while the larger scale NYC covering many consumers shows smoother changes.

The UK dataset shows a similar profile to the NYC dataset, with slightly higher means and extremums.

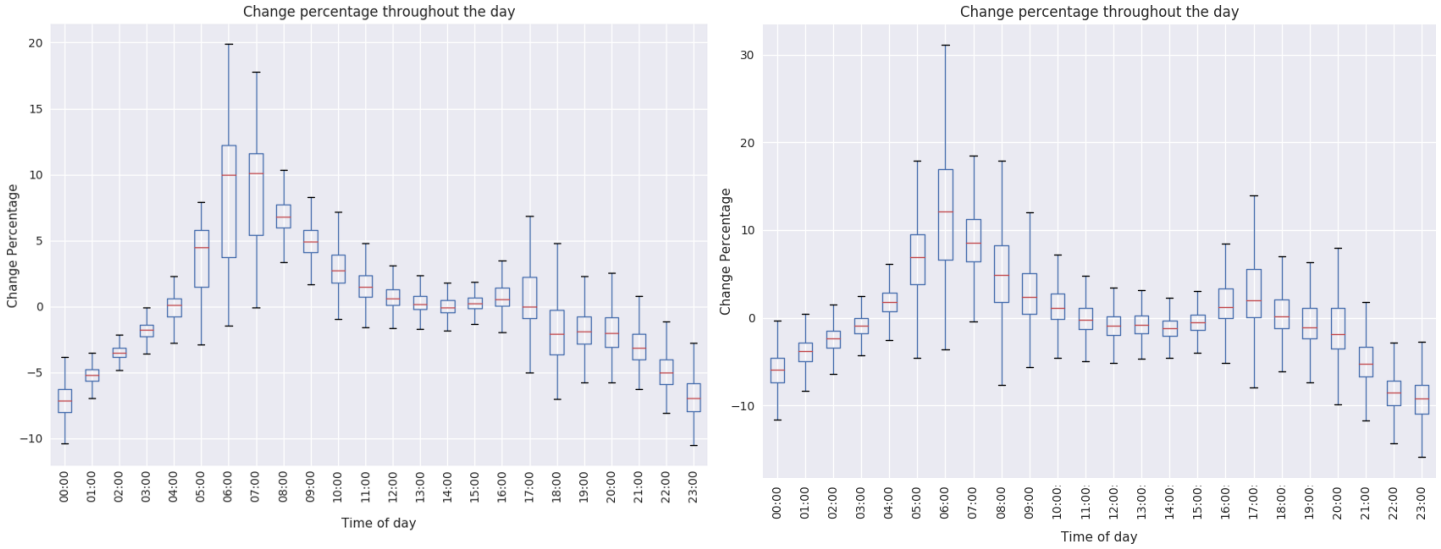


FIGURE 3.3: Hourly rate of change for the NYC and MHK datasets

This short analysis shows that these datasets have different dynamics, hence, forecasting performance may vary.

3.2 Weather datasets

NOAA's NCDC weather data

This database contains data from thousands of weather stations managed by the USAF[1] all around the world.

The data collection isn't straightforward but instructions can be found in Appendix A.

2 datasets were collected, one containing data from weathers stations within the UK (722 of them), similarly, one for the New York state (126 stations).

These contain a large number abnormal values, handled in Section 7.1.3.2.

3.2.1 Weather datasets analysis

In this section, we perform some analysis of the weather data (averaged over all stations by location). Since weather data is easily interpretable and commonplace, we can check if it meets our expectations. For example, we can verify that the peak temperature is greater in New York than in the UK and that summer temperatures are higher than winter temperatures.

[1] US Air Force.

Air temperature

In Figure 3.4, we can see that the UK temperatures are more temperate, ranging from 0 to 24°C (25^{th} and 75^{th} percentiles) while it ranges from -10 to 34°C. This corroborates with the hypothesis that the summer load profile could varies between the UK and NY due to air conditioning; yet it may not be the only factor.

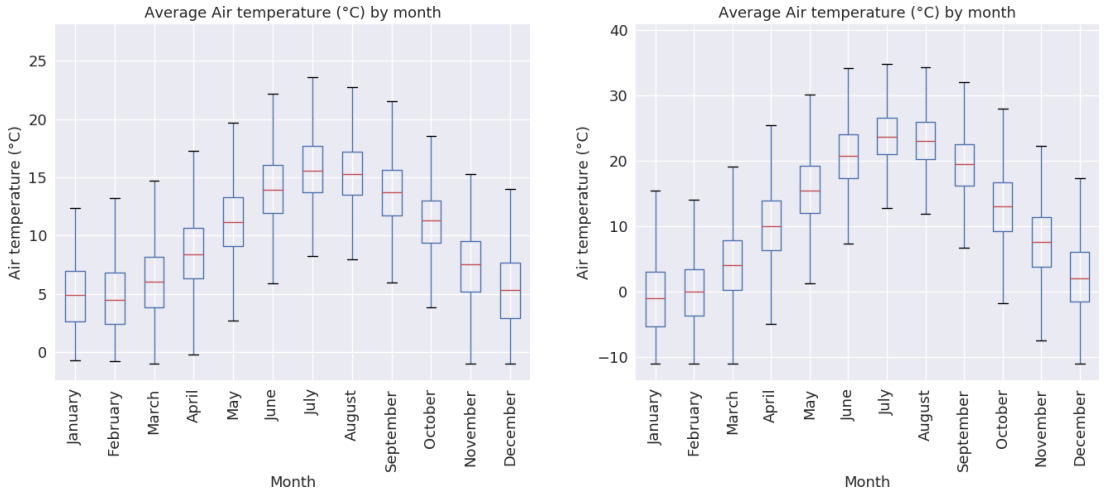


FIGURE 3.4: Monthly air temperature in the UK and New York state

Wind speed

Wind speed are slightly higher in the UK than in the New York state, but share a similar profile, at least when aggregated monthly.

Humidity

The humidity (%) tends to be higher in the UK, averaging around 85% compared to 70% in the New York state. Also, the humidity levels can drop much lower in the New York state with a 25 percentile of 20% humidity compared 43% in the UK.

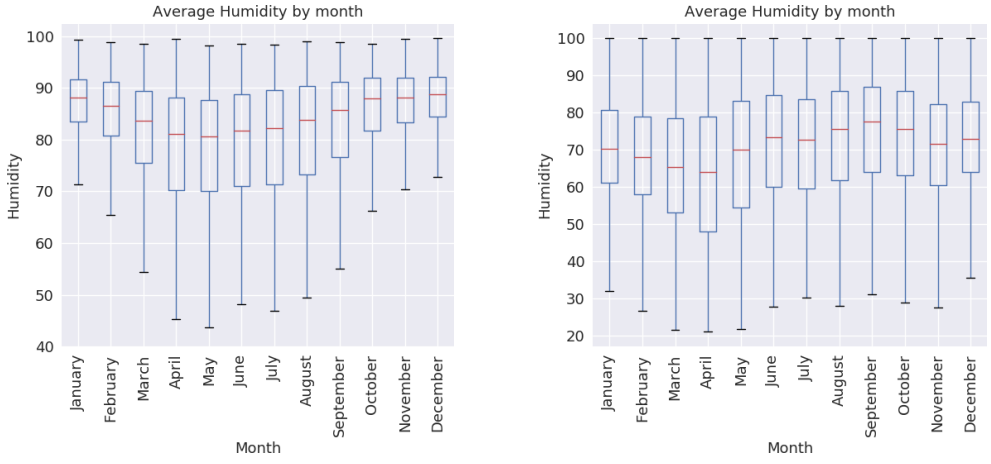


FIGURE 3.5: Monthly humidity in the UK and New York state

Chapter 4

Literature review

There exist various approaches for time series forecasting and in this section, we will highlight the most common ones in the context of electricity load forecasting.

4.1 Physics-based / hybrid methods

One possible way to address the problem - and maybe the most intuitive one - is to model the exact phenomena driving the time series of interest by formulating physics equations driving this electric consumption, including transmission losses and variability due to factors such as temperature. This would theoretically lead, if all relevant phenomena are modeled, to the most accurate model possible. However, noise is always present and as it is inherently random, this model would still not be able to create error-free forecasts.

To the best of my knowledge, no research paper on a purely physics-based model applied to electric forecasting currently exists. The main reason is the inherent complexity of such task, as one would have to somehow model the usage pattern of every electric device connected to the grid.

However, hybrid solutions have been attempted, for example, in Dong et al. [2016]'s work, data-driven models were combined with physical models. The experiments are conducted on a small number (4) of households with different types of insulation materials and use high-frequency data for one of the datasets (5 minutes interval). As pointed by the authors, this level of precision is often not practically available, but the smart-grid may be a solution to that problem.

Three models are concurrently used to produce the final forecast : two physics-based models alongside a data-driven model. The thermodynamics modelling is divided into 2 models, one responsible for Air Conditioning (AC) load (using high-frequency data), the other modelling building heat exchanges (using weather information). The data-driven model is tasked with the forecasting of all other appliances - for which the usage patterns are more complex than for the AC load which is strongly correlated with the temperature.

The results are compared to several data-driven models and show slightly better overall forecasting performance than the next three best performing models (Artificial Neural Networks (ANN), Support Vector Regression (SVR) and Least Square Support Vector Machine (LSSVM)). The other data-driven models forecasts showed significantly larger forecasting errors (Gaussian Mixture Model and Gaussian Process Regression). It is important to note that the proposed model was not tested against any Deep Learning based models and these are known to provide better forecasts than previous data-driven models such as Artificial Neural Networks.

4.2 Statistical methods

Statistical methods for time series analysis ('describing') and forecasting ('predicting') have been widely used. However, at least in the field of energy forecasting, they tend to be outshone by data-driven models such as Artificial Neural Networks (ANN), mostly due to their limited modelling complexity; and as discussed previously, electric load time series are usually very complex. ARIMA and Holt-Winters models are the most widely used statistical forecasting models, thus, are discussed below.

4.2.0.1 ARIMA

The Autoregressive Integrated Moving Average (ARIMA) model is one of the most commonly used statistical model for load forecasting, and more generally for non-stationary time series.

ARIMA models are a class of model typically denoted as "ARIMA(p,d,q)" where p, d and q respectively correspond to the order of the AR model, the degree of differencing and the order of the MA model.

This first step with this method is to make the time series "stationary", meaning that the series must be have a stable mean value (versus time) and consequently contain no trend or seasonality. . To remove these components, we can "difference" the time series, once to remove the trend (first difference, $d = 1$), by subtracting the previous observation from the current one. To remove remaining the trend (due to seasonality), we can difference the time series again (second-order differencing, $d = 2$). This operation corresponds to the Integral (I) term in "ARIMA".

The AR model in ARIMA is used to identify (linear) correlations between current and past values, therefore the output is related to the current input and the p (order) previous (differenced) values $(X'_{t-p}, ..., X'_{t-1})$. The output of such model is written as in Equation 4.1.

The MA model is very similar to the AR model but attempts to model the relationship between X_t and past noise ε_t instead of past time series values. It is written as in Equation 4.2

The full ARIMA model is expressed as in Equation 4.3.

$$X'_t = \mu + \sum_{i=1}^{p} \varphi_i X'_{t-i} + \varepsilon_t \qquad \text{AR model} \qquad (4.1)$$

$$X'_t = \mu + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} \qquad \text{MA model} \qquad (4.2)$$

$$X'_t = \mu + \sum_{i=1}^{p} \varphi_i X'_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} + \varepsilon_t \qquad \text{ARIMA model} \qquad (4.3)$$

where φ_i and θ_i are the AR and MA factors, μ is a mean of the series, X'_t and ε_t are the value of the (differenced) time series and noise at time step t.

The model parameters (factors) are identified using various statistical techniques as to give the best possible predictions but won't be discussed in this report.

4.2.0.2 Seasonal Holt-Winters

Seasonal Holt-Winters (SHW) also called triple exponential smoothing is a kind of smoothing method. Smoothing methods are applied to time series in order to reveal its underlying components (such as trend and seasonality).

The simplest smoothing method is to average all past data, the result becoming the forecast value. A similar method, named Moving Average [1] considers the forecast value to be the average of the last N data points only. A slightly improved version applies weights to every data point involved in the calculation of the forecast value (Weighted Moving Average). A special case of this method is the Single Exponential Smoothing method, with uses exponentially decaying weights for older data points, therefore, sometimes referred to as Exponential Weighted Moving Average (EWMA).

However, if any trend is present in the time series, the forecasting error (i.e. Mean Squared Error) using any of these methods will be large.

In 1957, Holt proposed an improved version of his Single Exponential Smoothing technique capable to handle time series containing a trend component, called Double Exponential Smoothing technique.

Later, Winters further improves it, adding seasonal handling capability : seasonal Triple Exponential Smoothing, sometimes called Holt-Winters.

The forecast value is computed as $\hat{y}_{x+m} = \ell_x + mb_x + s_{x-L+1+(m-1)modL}$ with ℓ_x, b_x and s_x equal to :

$$\ell_x = \alpha(y_x - s_{x-L}) + (1-\alpha)(\ell_{x-1} + b_{x-1}) \quad \text{level} \quad (4.4)$$

$$b_x = \beta(\ell_x - \ell_{x-1}) + (1-\beta)b_{x-1} \quad \text{trend} \quad (4.5)$$

$$s_x = \gamma(y_x - \ell_x) + (1-\gamma)s_{x-L} \quad \text{seasonal} \quad (4.6)$$

Variables α, β and γ are hyper-parameters with a large search space (3 decimal point precision results in a search space of 1 million combinations) : exhaustive search is therefore not a viable solution.

One can use trial-and-error or use a learning algorithm - similar to those described in later sections - in order to fit parameters to the data, in other words, find values of α, β and γ such that the forecasting error is minimized.

When applied to the problem of electric load forecasting, a variant of Holt-Winters capable to work with multiple seasonal components (such as week and seasons periodicity) is often used, called Double Seasonal Holt-Winters (DSHW).

[1] Not to be confused with the Moving-Average (MA) model.

4.3 Machine learning models

4.3.1 Machine learning introduction

Machine learning (ML) is a field of computer science gathering algorithms capable to give a system the ability to "learn". Learning in this context can be defined as improving performance on a given task through the exploration of data, without the computer being explicitly programmed. Machine learning is a subfield of Artificial Intelligence (AI) and is currently a very popular way of solving many problems, from computer vision and speech recognition to spam filtering and finance. It is also widely applied to our problem of time series forecasting.

There are two main categories of learning procedures in Machine Learning (ML), supervised and unsupervised learning. In a supervised environment, the algorithms gets to know the expected output (called label) for a given input and tunes its internal variables (i.e. a neuron's weight in a Artificial Neural Network) so that the model's output gets closer to the expect output. Supervised learning can be further divided into sub-categories and the above definition actually corresponds to the "active learning" scheme and is the most prevalent type of supervised learning. All ML models explored in this dissertation fall into that category.

On the other end, in an unsupervised learning environment, no labels are provided and the model has to find structures in the data on its own. This scheme won't be explored in this project.

Another important aspect of ML we need to cover is the difference between a regression and a classification problem. When dealing with classification problems, the goal is to classify an input as belonging to a given class (output). A common example is to classify data points represented by a vector of attributes such as height and weight into two output classes male and female. The attributes can be discrete or continuous while the output is discrete (classes). First, the model is trained using pairs of input-output data (called training set) and is then is evaluated against a test set where the output class is not known and the model is tasked to predict it. The output is usually a discrete probability distribution and the most likely class is chosen as final output.

Conversely, when solving a regression problem, like time series forecasting, the output is continuous.

When designing and training any machine learning model, one must not only look at the performance of their model on the training set but also on the test. The main reason being that a model can be underfitting or overfitting the training data. A overfitting model behaves in such as way that its parameters as perfectly tuned to fit the training data (including noise) to the point where the testing error will be far greater than the training error at which point we say that the model is not capable to generalize. Some level of discrepancy is expected, but this scenario is the signature of an overfitting model. This can occur when the model complexity is too large compared to the data complexity. On the other hand, when a model's complexity is too small, the model is not capable to capture the underlying structure of the data. Both underfitting and overfitting lead to poor performance and mitigations are generally model specific. Underfitting and overfitting models are shown on Figure 4.1.

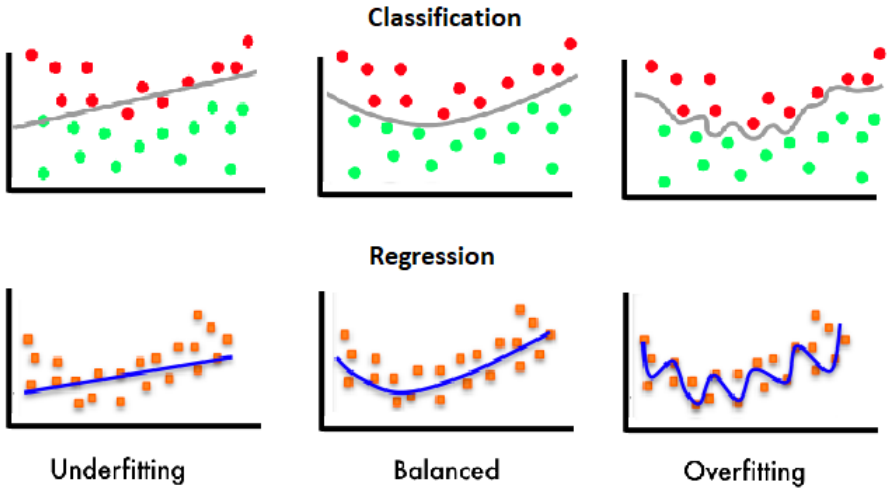


FIGURE 4.1: Overfitting/underfitting on classification and regression tasks.

A commonly used metric to assess the performance of a machine learning classification model on a given dataset is the "accuracy". It is a simple metric that gives an overall picture of a model's performance and is computed as the number of correctly predicted instances of the test set divided by the count of test instances.

In the case of regression problems, continuous metrics are used such as the Mean Absolute Percentage Error (MAPE) and will be discussed in Section 4.5 .

4.3.2 Support Vector Regression (SVR)

Support Vector Machines are a class of supervised algorithms, with the original algorithm, now called Linear SVM being suitable for binary linear classification. Many versions now exist, modifying the original capabilities of the original SVM algorithm and some of them will be introduced. The linear SVM is a non-probabilistic linear binary classification algorithm that aims at dividing (classification) 2 classes (binary) by a hyperplane (linear).

A hyperplane is a geometrical construct that has one dimension less than the space in which it resides, for example, a hyperplane in a 1D space is a point and a line in 2D space. If the training data is linearly separable, the maximum-margin hyperplane perfectly separates the two classes and is located as far as possible from the closest instance of both classes. The support vectors are hyperplanes such that they create a space where no class instances lie and represent what is called the margin. Therefore, the maximum-margin hyperplane is halfway between these hyperplanes.

In case the dataset is not linearly separable as most non-artificial datasets are, we cannot determine a hard-margin. Instead, we use a soft-margin which allows for some data points (errors) to lie within the margin. There is a trade-off between low training error (few data points lie within the margin) and generalization capability and is represented by the regularization parameter C or λ. A hard-margin has $lim_{C \to \infty}$, meaning it tolerates no errors in the training set. However, using a hard-margin in presence of outliers and noisy data can make a SVM overfit (small margin) or not capable to find a maximum-margin hyperplane if the dataset is not linearly separable.

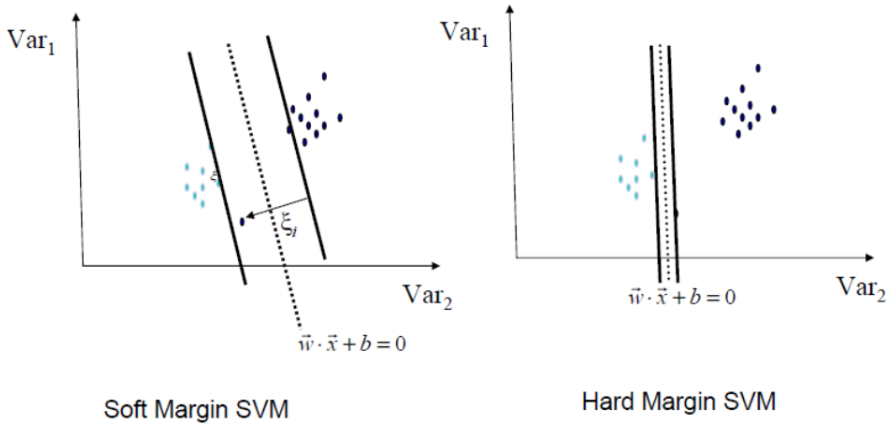


FIGURE 4.2: SVM binary linear classification using soft and hard margins. (of-Information-and Computer-Sciences)

As we can see on 4.2, an outlier from class "black" is close to the instances of the other class "blue". Using a hard-margin, the SVM still finds a maximum-margin hyperplane but that margin is very small, leading to overfitting as a testing data point belonging to the "blue" class but slightly to the right of the training data points of the "blue" class would be classified as "black". On the other-end, the soft margin offers a much larger margin as it tolerates some errors in the training set, in turn making the SVM able to better generalize.

An important variant of the SVM is the Support Vector Regression (SVR) method (Drucker et al. [1997]), a modification of the SVM suitable for solving regression problems.
A variant of the SVR, proposed by Suykens and Vandewalle [1999] called Least Squares Support Vector Machine (LS-SVM) is especially suitable for time series such as our problem of electric load forecasting.

4.3.3 Artificial Neural Networks (ANN) based models

Artificial Neural Networks (ANN) are mathematical models partially based on our understanding of the human brain. The following explanations are valid for most models, but some may modify some specific behaviours.

ANNs are composed of elementary units called artificial neurons, usually arranged in layers as shown on Figure 4.3 :

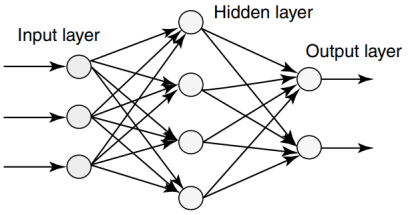


FIGURE 4.3: Artificial Neural Network (Abraham [2005])

Despite their recent breakthrough in various fields, they are not new by any means and date back from the 1950's. Various factors lead Artificial Neural Networks to become very popular Machine Learning models; the two most important aspects are the availability of data, often referred to as Big Data, and the exponential increase in computing power enabling for more complex models.

Weights

Neurons are connected via connections called weights representing the importance of a given input with regards to the other inputs of the target neuron. Weights are usually floating point values and can be regarded as the connection strength between two neurons. These weights are adjusted during training in a way similar to a process that occurs in the brain, called synaptic plasticity. Weights and neurons are typically arranged in layers and the most common ANN is the feedforward Multi-Layer Perceptron (MLP) with 2 layers : a hidden layer and an output layer. Note that the input "layer" isn't composed of neurons but only of placeholders for user data. This data is then fed into the hidden layer's neurons.

Different types of layers exist, the most common one being the "fully-connected" / "dense" layer where the input signal of every neuron from the previous layer is fed into every neuron of the next layer.

Neurons

There exist multiple types of neurons, but all follow a two-step process in order to compute an output given some inputs. First, each input signal is multiplied by its corresponding weight, then the sum goes through a non-linear activation function.

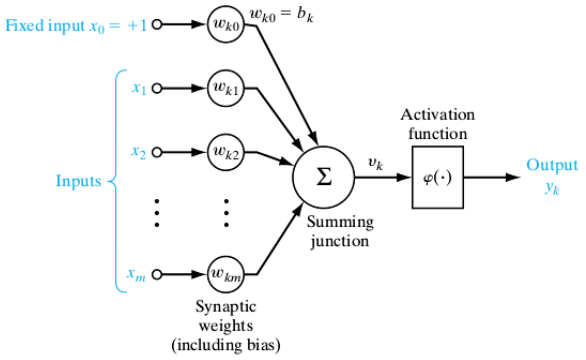


FIGURE 4.4: Artificial Neuron ([Haykin, 2009, p.11])

The output of a neuron k neuron is given in equation 4.7.

$$Y_k = \varphi\left(\sum_{j=0}^{m} w_{kj}x_j + b_k\right) \qquad \text{Artificial Neuron output} \qquad (4.7)$$

ϕ : Activation function

w_{kj} : Weight from neuron j (layer $n-1$) to neuron k (layer n)

b_k : Bias weight in layer k

x_i : i^{th} input signal from the previous layer $n-1$

A bias neuron is often added to every layer of a neural network (except for the output layer) and is used to shift the input of neurons, allowing for more flexibility in the learned function.

Activation functions

An activation function transforms the neuron summation into the neuron output, usually in a non-linear way in order to be able to fit non-linear datasets (i.e. vast majority of real-world dataset).

Many kinds of activation functions exist, the most popular being the logistic sigmoid activation function (4.8) and the Rectified Linear Unit (4.9). Another commonly used activation function, very similar to the sigmoid function is the hyperbolic tangent function (4.10).

$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \text{Sigmoid} \qquad (4.8)$$

$$f(x) = \max(0, x) \qquad \text{ReLU} \qquad (4.9)$$

$$\tanh(x) = 2\sigma(2x) - 1 \qquad \text{Tanh} \qquad (4.10)$$

ReLU is the most commonly used activation function in deep learning as it helps in solving one of the challenges faced when training them, namely the "vanishing gradient problem", discussed in more details later, in Section 4.3.3.1. The ReLU function and its derivate are also less computationally expensive to compute compared to the sigmoid and hyperbolic tangent functions.

(A) ReLU (B) Sigmoid

FIGURE 4.5: ReLU and sigmoid activation functions and their derivatives Hao [2017]

It should be noted that the activation function used for the output layer of an ANN usually differs from the ones used in the hidden layers. In case of a regression problem, a linear activation function (equivalent to not using an activation function) is typically used. For binary classification tasks, a sigmoid function can be used. For multi-class classification the generalization of the sigmoid function is commonly used : the softmax function.

The choice of activation function has a measurable effect on training time and network performance as seen on Figure 4.6 where an MLP was used for day-ahead load forecasting using a hourly load dataset.

Activation function of output layer	Activation function of hidden layer								
	Hyperbolic sigmoid			Hyperbolic tangent			Linear		
	(A)	(B)	(C)	(A)	(B)	(C)	(A)	(B)	(C)
Hyperbolic sigmoid	2.030	2.048	2.625	1.510	1.621	1.817	1.788	1.850	2.091
Hyperbolic tangent	1.671	1.737	2.042	1.383	1.482	1.749	1.900	1.987	2.200
Linear	1.603	1.691	1.903	1.390	1.522	1.747	1.936	2.023	2.194

FIGURE 4.6: Effect of various activation function on training (A), validation (B) and testing (C) error (Tsekouras et al. [2015])

Keep in mind that; as often in Machine Learning, these results won't necessarily be valid for every model and on every dataset.

Learning process

The weights of the networks define its capabilities, such as predicting the price of a house given various metrics of the house. In order to obtain a network capable to output any meaningful results, it needs to be trained. However, in order to be able to assess a model's performance, we need to define an error metric, more formally called cost function.

A typical neural network architecture work-flow is shown in Figure 4.7. This graph can easily be modified to fit any machine learning algorithm by changing the "Neural network initialization" step to a more generic "Model initialization" step.

At this point, the reader may wonder why it would not be possible to find the optimal weights analytically, which would make the process much more straightforward and yield optimal performance in a non-deterministic manner. Unfortunately, it is very difficult, if not impossible to solve systems containing thousands of parameters (network weights), hence the need for a more suitable method, discussed in this section.

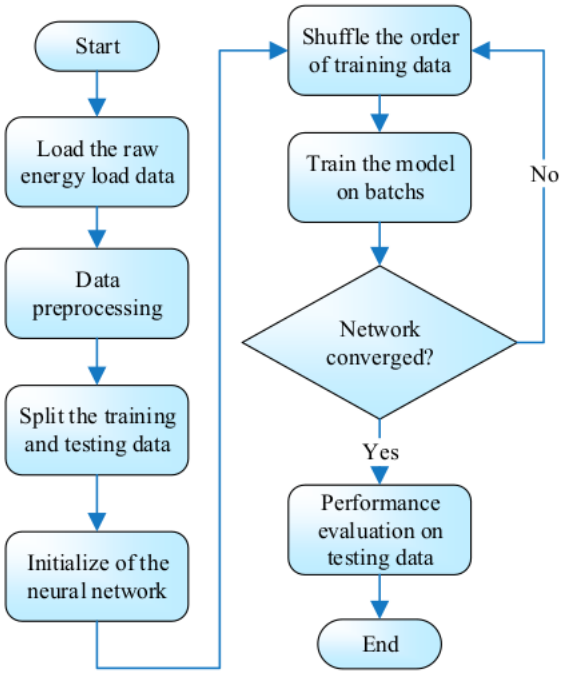


FIGURE 4.7: Neural network work flow (Kuo and Huang [2018])

Cost function

The goal of any machine learning model is to minimize this cost function, a scalar, regardless of what it tries to model. The terms "loss" and "cost" are often used interchangeably, but formally speaking, a cost function is made up of a loss function and an optional regularization factor which we will explore later.

$$J(\theta) = \frac{1}{2N}\left[\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2 + \lambda R(\theta)\right] \qquad \text{Cost function} \qquad (4.11)$$

$\hat{Y}_i$: Predicted output value for the i^{th} element

Y_i : True output value for the i^{th} element

θ : Model parameters

λ : Regularization factor

R : Regularization term (discussed later)

N : Number of samples in the dataset

M : Number of model parameters

There exist a variety of loss functions and their usage depend on the problem at hand. We actually already presented an error function that could be used for classification tasks in Section 4.3.1 : the "accuracy" metric. However, ANNs are probabilistic classifiers, meaning that they output a probability for every given class, yet, the accuracy metric only rewards instances that are correctly classified according to a given threshold (0.5 for example) and gives no insight into the details of the predictions. For example, a model classifications such as (0.45, 0.55) where the target is (1,0) would have the same accuracy as a model outputting (0.05, 0.95). Therefore, the accuracy isn't a suitable cost function in most cases.

In a supervised-learning scenario, commonly used loss functions are the quadratic cost also known as Mean Squared Error (MSE) or L2 Norm (4.12), the Mean Absolute Error (MAE), also called L1 norm (4.13) and the cross-entropy cost (4.14).

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \qquad \text{Quadratic Cost - MSE} \quad (4.12)$$

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left|Y_i - \hat{Y}_i\right| \qquad \text{MAE} \quad (4.13)$$

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\left[Y_i \log_{10}(\hat{Y}_i) + (1 - Y_i)\log_{10}(1 - \hat{Y}_i)\right] \qquad \text{Cross-Entropy} \quad (4.14)$$

Now that we know how good our network is, we need to find a way to make it produce relevant outputs. The naive approach is to randomly select weights values and keep the combination of weights giving the smallest output error, this is called random search. Due to the very large search space (if weights value precision is limited, otherwise the search space is infinite), this is not a suitable approach.

An naive example of a cost function is shown on Figure 4.8.

Weights Optimization

Instead, optimization algorithms are typically used to minimize the cost function and therefore improve our model's predictions.

The most common algorithm is called batch gradient descent in which the gradient of the cost is calculated for the entire training dataset. The gradient ∇ represents the 'uphill slope' (gradient ascent) of the cost function at a given point. Since we want to minimize the error, we update the model's parameters in the opposite direction of the gradient using equation 4.15.

$$\theta := \theta - \eta \nabla J(\theta) \qquad \text{Batch gradient descent} \qquad (4.15)$$

θ : Model parameters

η : Learning rate

∇ : Gradient

$J(\theta)$: Cost function

A variant, called stochastic gradient descent (SGD) updates the weights after every training sample instead of using the averaged errors of the whole training dataset. This is called "on-line" training and effectively enable the network to train as it sees new data and not only updates all its weights after all the training set is passed through it.

Stochastic gradient descent is computationally expensive since the gradient is computed for every training sample. However as the gradient is calculated on noisier data (single training examples), it is less likely to get stuck in local minimums (meaning that the gradient is very small). On the contrary, batch gradient descent is very fast (single computation) but is more susceptible to the aforementioned issue.

Therefore, a third algorithm called mini-batch gradient descent is often used and combines the advantages of both previous methods. The weights are updated every N samples, N being the mini-batch size. The average cost on the mini-batch is typically used as an approximation of the real cost function on each sample. As such, mini-batch gradient descent offers a good balance between the robustness of SGD and the computational efficiency of batch gradient descent.

Learning rate

The learning rate η is a parameter that affects the update size after each iteration of gradient descent. The learning rate is directly proportional to the speed at which the stochastic gradient descent algorithms converges. However, a learning rate too high can hinder convergence or even cause the process to diverge.

An ideal cost function (versus a single network parameter) is shown in Figure 4.8 along with the effects of learning using various learning rates. The red line shows a constant high learning rate where the stochastic gradient descent algorithm doesn't reach the global minimum, which diverges due to the learning rate being too high. The blue arrows represent a suitable learning rate. An alternative strategy is to use an adaptive scheme (green line) where the learning rate is scaled according to various parameters such as the local gradient. This helps in making gradient converge faster while avoiding any divergence. Often times, a trial-and-error approach for a suitable learning rate is used as it usually provides great convergence speed with no added system complexity.

While the learning rate is a simple looking parameter, it plays a large role in neural networks, especially deep neural networks. A lot of research is made to develop alternatives to the SGD algorithm that better prevent the algorithm from getting trapped in local minima and speed up convergence.

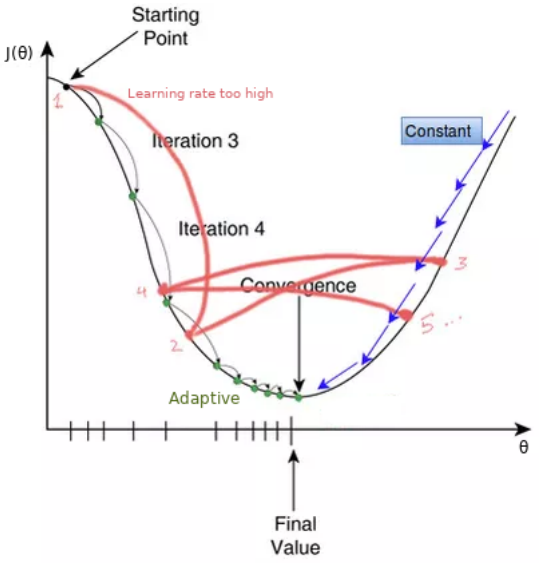


FIGURE 4.8: Effect of learning rate on the gradient descent algorithm

Note that the case studied in Figure 4.8 isn't realistic as the cost function over model parameters is usually not convex but instead has many saddle points (derivative equals zero), making the process of convergence to the global minimum more challenging.

It is important to note that the gradient descent algorithm doesn't define how to compute the gradient of the loss function. The back-propagation algorithm is often used in neural networks as it provides an efficient way to calculate the gradient at every node (neuron) of the network.

Gradient descent variants

While SGD is a commonly used optimization function, several other optimizer functions based on SGD exist such as Adam (Kingma and Ba [2014]), in which the learning rate is dynamically changed, along with other algorithmic modifications when compared to Stochastic Gradient Descent. It is based on the Adaptive Gradient algorithm (AdaGrad) and Root Mean Square Propagation algorithm (RMSProp) by combining the advantages of both.

Sebastian Ruder compares many modern optimization algorithms in his paper (Ruder [2016a]), also available on his website (Ruder [2016b]) with added animated graphs.

Now that we have defined the basic building blocks of a Neural Networks, we will explore 2 of the most common ANN design difficulties.

Model structure

The first one is related to the architecture of the network. How many layers should our network have ? How many neurons by layers ? Unfortunately, there are very few design rules for designing Artificial Neural Networks. The only known variables are the numbers of neurons in the input layer as it corresponds to the dimensionality of the input data. Similarly, the number of neurons in the output layer is problem-dependent and is typically straightforward to determine. However, there exist no hard-rule regarding the ideal number of hidden layers and number of neuron per hidden layers. In some kinds of ANNs, the number of hidden neurons tends to decrease the deeper a layer is in the network (towards the output layer), but this still is merely a rule-of-thumb and is very problem and architecture dependent. Therefore, ANN design are either by experimenting with various architectures (see Figure 4.9) or by using bio-inspired exploration techniques such as Particle Swarm Optimization (PSO) used by Bashir and El-Hawary [2009] for example. [1]

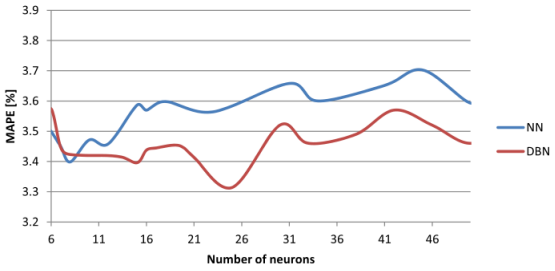


FIGURE 4.9: Validation set performance vs number of neurons per hidden layer (Dedinec et al. [2016])

[1] PSO (Eberhart and Kennedy [1995]) is an iterative population-based optimization technique in which individuals concurrently search for a common objective, such a minimizing a function with a behaviour similar to the flocking of birds.

Overfitting

The other important problem that often makes Neural Networks difficult to exploit is over-fitting. As briefly discussed in Section 4.3.1, we say that a model is overfitting when its accuracy on the training set is much greater than on the testing set. This can be regarded as the model memorizing every training sample without really modelling the underlying relationships, therefore performing poorly on unseen data.

The development of techniques to reduce overfitting is a very active research field but the main factor is the model's complexity with regards to the quantity and complexity of input data. Solutions to over-fitting can be model-dependent and in the case of ANNs, a few mitigation techniques are commonly used.

Regularization

Techniques used to reduce overfitting are called "regularization" strategies. The common goal is to reduce the generalization error (test set) even at the expense of increasing the training error, by altering the training process of the model.

The most common ones are :

- Model simplification

  Since overfitting comes from the difference between training data and model complexities, simplifying the model is a very effective solution in some cases (e.g : removing hidden layers, decreasing the number of neurons in hidden layers).

- Training set size increase

  This can be done by collecting more data or augmenting it. Augmenting a dataset means artificially generating new data examples from existing ones. An example with images would be to rotate or warp training images slightly. Larger training sets reduce overfitting by forcing the model to discover underlying data relationships and can give the upper-hand to a more complex model compared to a simpler one (see Figure 4.10).

- Cost function modifications

  Modifications to the cost function (Equation 4.11 - R term) prevent network weights from getting too large (which reduces generalization capabilities). A commonly used regularization technique is L2 regularization. It adds a penalty term ($R = \|\theta\|^2 = \sum_{i=1}^{M} \theta^2$) to the cost function. L1 and Max-norm regularization can also be used. The reason it works is because we are trying to minimize the cost function, therefore minimizing the penalty term, thus the weights.

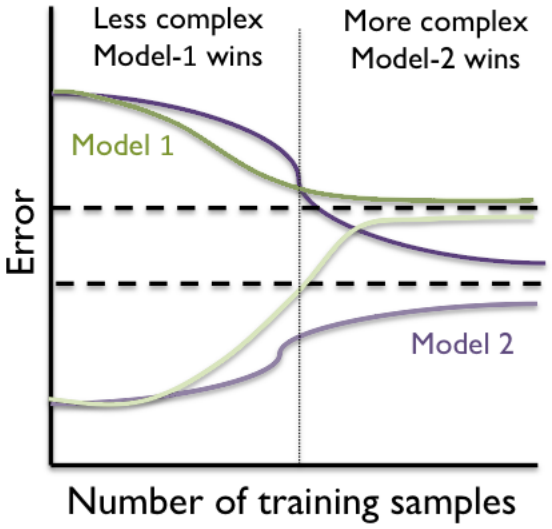


FIGURE 4.10: Testing (darker) and training (lighter) error vs training dataset size (Mathur [2012])

- Early Stopping

  The training process is stopped as soon as the accuracy on a 3^{rd} dataset, called validation set starts decreasing when training. This prevents the model from memorizing the training examples.

- Dropout (Srivastava et al. [2014])

  During training, random neurons are ignored, as if they were not part of the model. This prevents the model from strongly relying on specific neurons, forcing multiple internal representations of the data to be learned. Ryu et al. [2016], for example, used such technique.

Conclusion

As one might expect, every characteristic of an ANN can have an impact on its efficiency or applicability to a given problem and are therefore intensely researched. This results in the existence of many variations that are too numerous to cover in this dissertation.

Additionally, ANNs have many more complex variants that find success in specific application fields. In the next sections, we will discuss the concept of "deep learning" and introduce architectures that proved to be suitable for solving the problem of electric load forecasting.

4.3.3.1 Deep Learning

Deep learning is a very active subfield of machine learning and currently represents the state of the art for solving many problems from computer vision (such as Redmon et al. [2015]'s work on object detection) to natural language generation (Wang et al. [2017]) and mastering very complex board games (Silver et al. [2017]). Despite its recent exposure to the general public, deep learning isn't new as we can witness in Figure 4.11. However, while it offers impressive performance, including in the field of electric load forecasting, it carries many challenges.

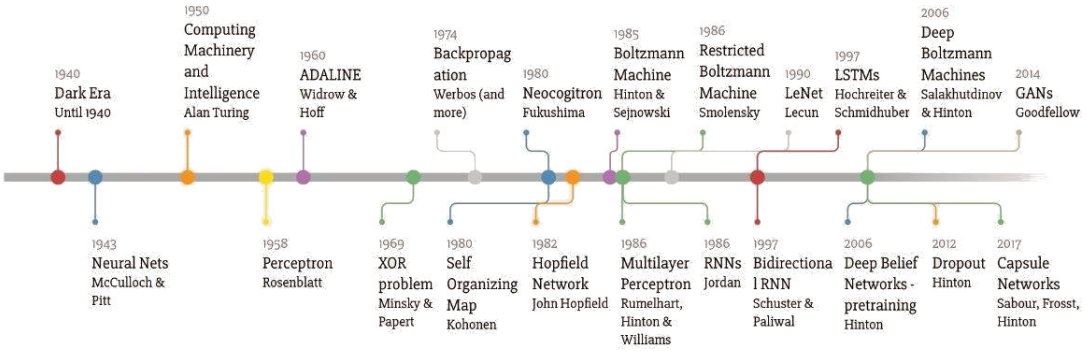


FIGURE 4.11: Deep Learning timeline (by Favio Vazquez)

Main features

Deep learning is a class of algorithms typically based on artificial neural networks described above. However, these deep networks contain more than a single hidden layer, improving their feature extraction capabilities. In the case of image classification, each layer can be seen as an abstraction used by the deeper layer, each extracting specific image features, these features becoming more and more complex the deeper the layer is in the network. These networks may be feed-forward, meaning that information moves unidirectionally from the input layer to the output, without cycles within layers. These networks are typically called deep multilayer perceptron and are part of the broad family of Deep Neural Networks (DNN). Some networks have cycles within layers, these are called Recurrent Neural Networks (RNN) while others are vastly different such as Convolutional Neural Networks. These networks, among others, will be described in the following sections before a short introduction to the main challenges faced when training deep networks.

Challenges

Unfortunately, this superior level of performance comes at a cost, mostly training complexity. Some of the biggest challenges to overcome are overfitting and various training issues including long training time. Overfitting is a concept we described in the above section along with some possible mitigations. Deep networks are more susceptible to overfitting due to their increased complexity and thus, are capable to "remember" entire training samples, therefore, providing low generalization capabilities as seen on Figure 4.12. Common mitigations techniques have already been presented in Section 4.3.1.

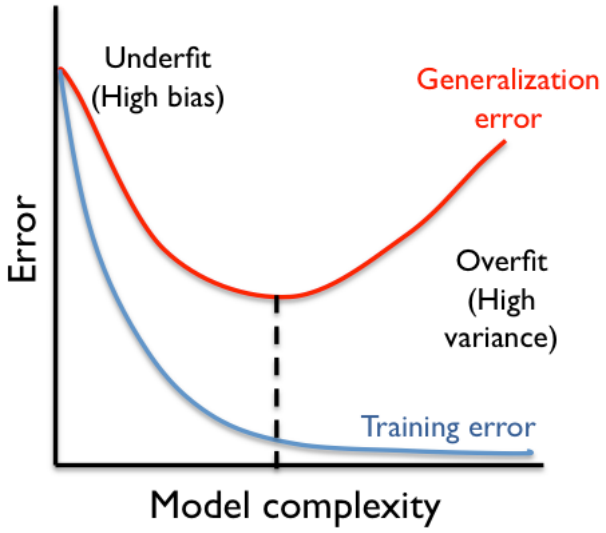


FIGURE 4.12: Training vs testing accuracy (Mathur [2012])

Training computational cost comes from the fact that deep networks can have tens of thousands of parameters to optimize (e.g. weights) thus requiring many forward and back propagation passes to train. Another factor is related to the computation of the gradient in the early layers of the network. The back-propagation algorithm computes the gradient of the error at a given layer. This gradient tends to become very small (vanishing) when computed for earlier layers, as these are computed using the chain rule thus multiplying small numbers due to the sigmoid activation function output being in the [0,1] range. The vanishing gradient problem, therefore, causes weight to be updated (trained) very slowly.

A commonly used method to alleviate this problem is to use ReLU activation functions (output $[0, \infty]$) in place of sigmoid in the hidden layers.

Other solutions include the use of LSTM networks (Hochreiter and Schmidhuber [1997]), especially when dealing with RNNs or using alternate training methods such as the one typically used for Deep Belief Networks which consists in training layers by layers (Hinton et al. [2006]) instead of training the whole network at once.

The increase of raw computational power also helps by simply performing many more weight updates in a given period.

All these problems remain under active research and some were responsible for the low prevalence of deep learning previously, showing how problematic these can be.

4.3.3.2 Recurrent Neural Networks (RNN)

Recurrent neural networks are a class of ANNs where feedback connections are added to the network. This gives RNNs an "internal state", some memory of past data, enabling the network to model time-based relationships between examples. This is a property that fit well our problem since most time series have strong auto-correlation.

Another consequence is input size flexibility. While on a non-RNN, input data size is fixed (number of input neurons), data can be fed in sequence and therefore have variable size. Both of these properties led RNN to have great success in various field such as speech recognition.

Fully-recurrent RNNs are the most basic type of RNN where the output from the previous time step is fed back as input along with the input data of the current time step.

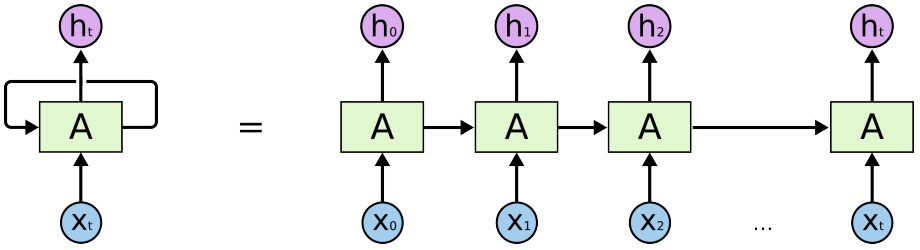


FIGURE 4.13: Fully-recurrent RNN unfolded over time (Olah [2015])

The type of RNN we are mainly interested in is called a Long Short-Term Memory neural network (LSTM - Hochreiter and Schmidhuber [1997]). These networks replace the usual feed-forward ANN structure with modules where data is no longer simply forward-propagated through the network. Instead, data undergoes multiple parallel operations as described in Figure 4.14.

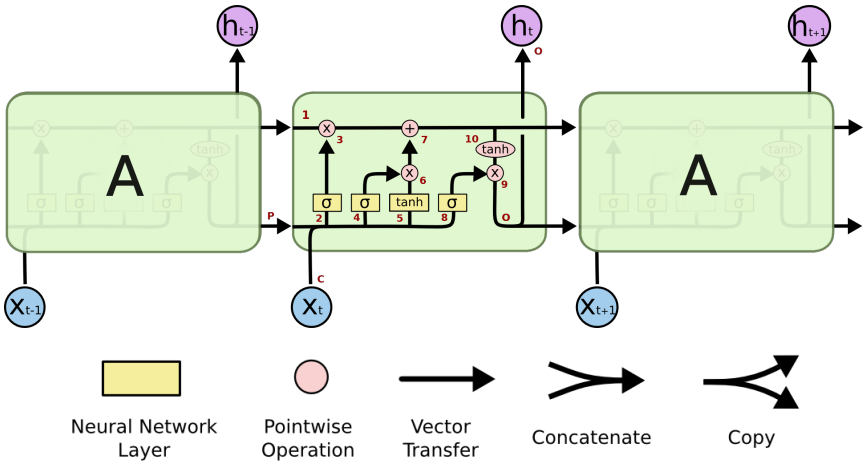


FIGURE 4.14: LSTM module (Olah [2015])

We can identify several components : a state cell, an input and output gate along with a forget gate. The top line (1) carries the internal cell state from the previous time step to the next. The leftmost layer is the forget gate layer (2) and output a value between 0 and 1 based on the current data (C) input and previous model's output (P). This value then determines how much of the previous cell state is kept for the current and subsequent computations. (3) Concurrently, the same information is carried through layer (4), called input gate layer determining which values from the cell state will be updated. Layer (5) builds candidate values to be stored in the cell state. The results of these 2 cells are multiplied (6) and added (7) to the remainder of the old cell state. Input data goes through layer (8) which ultimately determines the output (O) of the model by multiplying (9) its output by the filtered (10) cell state.

Curious readers would probably appreciate Olah's article for additional details and variants of LSTMs.

Note that by contrast, a basic RNN has a single hidden layer (similar to ANNs described in Section 4.3.3), with its inputs being the previous network output along the current data input.

As many deep learning models, RNNs models suffer from the vanishing gradient problem. This is caused by the gradient's derivative getting smaller and smaller the deeper the layer is in the network (starting from the output). This leads to very slow training, however, LSTMs were designed with that issue in mind and do not suffer from it.

4.3.3.3 Convolutional Neural Networks (CNN)

Convolutional neural networks (CNN) are often used with high dimensionality input data such as images. These networks can become extraordinarily deep and complex as demonstrated by the Inception models (Inception v4 - Szegedy et al. [2016] - Figure 4.15). The network uses the early layers to detect low-level features such as objects edges, these features are then used by deeper layers to detect more complex structures. This is akin to how the brain's visual cortex works; a related paper published by Hubel and Wiesel [1962], showed that cats have "simple" and "complex" cells capable to recognize different visual features in a similar way. (See : Video).

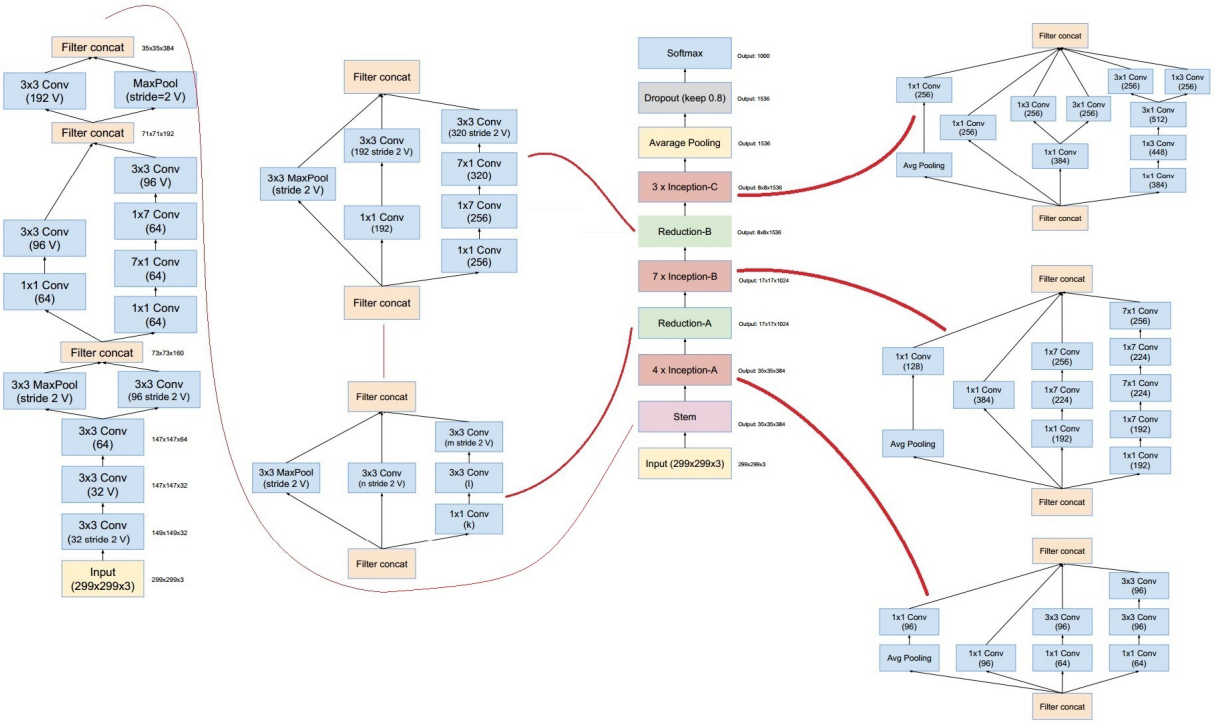


FIGURE 4.15: Inception V4 architecture (Szegedy et al. [2016])

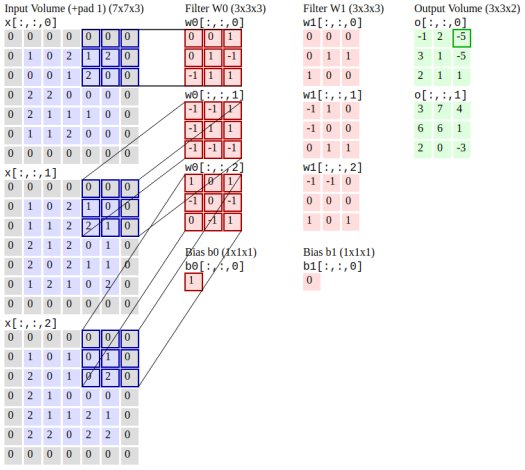
The following descriptions will be using of an image-based problem such as object detection.

Indeed, CNNs were built from the ground up with these types of problems in mind and are simpler to explain considering a similar task.

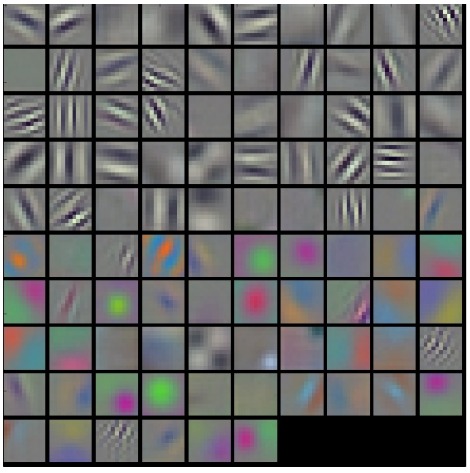
CNNs have 4 types of layers :

- Convolutional layer (CONV)

  These consist of a set of small filters (typically 2 dimensional) with shared trainable weights. These filters are convolved with the image (input image or output of previous layers) and can detect certain features. Their output is called an activation or feature map and gradually represents the presence of not of higher and higher level features (from edges to cat for example).



(A) Convolution process in a convolutional layer (Stanford-University [2017])



(B) Image representation of the first convolutional layer of a trained CNN (Krizhevsky et al. [2012])

FIGURE 4.16: Convolutional layer visualizations

On Figure 4.16a, a step of the convolution process is shown. The input volumes (image channels) are convoluted with two banks of filters W0 and W1. The sum of both convolutions form the two output volumes (activation maps).

- ReLu layers (RELU)

  The input of such layer is passed through the Rectified Linear Unit activation function (element-wise) and these are typically used after a CONV layer to introduce non-linearity.

- Pooling layers (POOL)

  Pooling layers perform down-sampling on the input data hence reducing data dimensionality. This, in turn, decreases the numbers of parameters, making the model faster to train and less susceptible to overfitting. These layers have three parameters : filter size, stride, operation type and padding type.

The operation typically is "max" or "average", resulting in so-called max-pooling and average pooling. Depending on filter size, stride and input volume size, padding around the image can be necessary. Zero-padding is typically used as in Figure 4.16a.

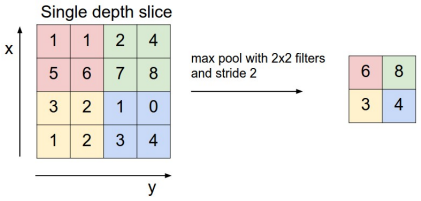


FIGURE 4.17: Pooling layer (Stanford-University [2017])

- Fully-connected layers (FC) / Dense layers

  This layer is identical to those we find in "standard" ANNs and has been described in 4.3.3. A fully-connected layer is placed at the end of a CNN and gives the various output classes detected by the network.

These layers can arranged in a repeating pattern such as 3 blocks of [CONV-RELU-CONV-RELU-POOL] layers followed by a fully-connected layer to obtain the final output.

An example of a CNN architecture can be seen in Figure 4.18.

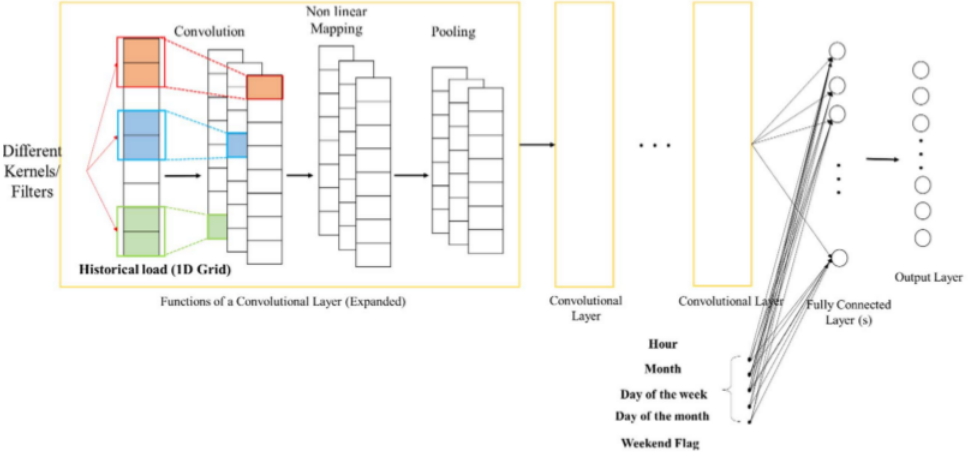


FIGURE 4.18: CNN architecture (Amarasinghe et al. [2017])

4.3.3.4 Deep Belief Networks (DBN)

Deep Belief Networks are made up of simple networks such as Restricted Boltzmann Machines (RBM) or auto-encoders. As opposed to traditional ANN, the DBN training procedure doesn't use the back-propagation algorithm. Instead, it consists of two steps, proposed by Hinton et al. [2006]. The first step is called "pre-training" : RBMs are trained in a greedy unsupervised way. "Greedy" means that the network is not trained as a whole, but layer (RBM) by layer. Once the first RBM is trained to reconstruct its own input, another is stacked on top of it, and the hidden units of the previous RBM act as the input layer of the next RBM. The second step is to "fine-tune" the network using labelled data (supervised learning). Their architecture is identical to those of Multi-Layer Perceptron described in Section 4.3.3. The "pre-training" is the crucial step as it acts as a weight initialization procedure for the network, greatly reducing the need for labelled data. On top of that, DBNs have shown to provide good performance as we will explore in section 4.6.

Due to their short supervised learning period with fewer epochs and lower weight

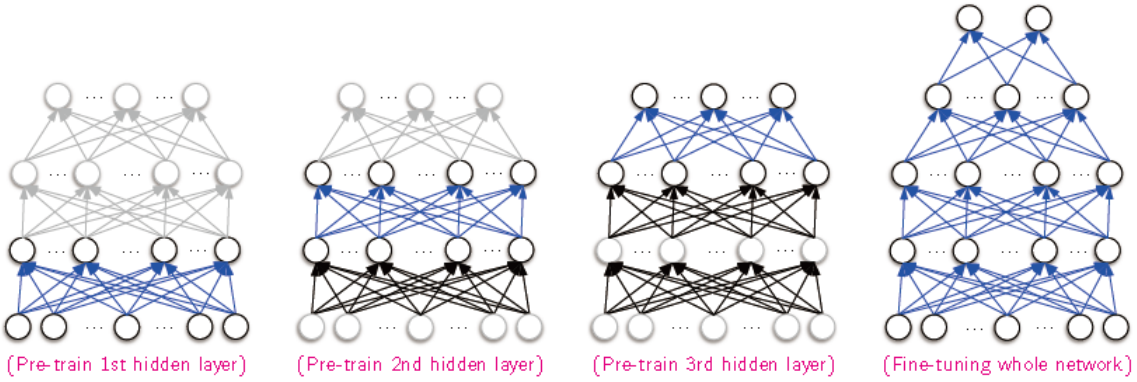


FIGURE 4.19: DBN training process (Joohyoung [2015])

Restricted Boltzmann Machine (RBM)

Restricted Boltzmann Machines (RBM), proposed by Smolensky [1986] are a variant of Boltzmann Machines (Ackley et al. [1985]). The restriction comes from the fact that neurons cannot have connections with other neurons of the same layer. Their architecture is similar to that of an MLP (shallow feed-forward network), more formally, RBMS are symmetrical (i.e. fully-connected) bipartite (i.e. 2 layers) graphs. RBMs learn the probability distribution of their inputs (generative model) and can be trained in a supervised or unsupervised (as in DBNs) manner. As such, they can be applied to various tasks such as dimensionality reduction, classification and feature learning (as in DBNs).

4.4 Input data

Before being able to compare the most relevant scientific papers, we must cover an important aspect of any machine learning model : the input data.

In this age of "Big Data", one might think that it is enough to gather large quantities of data to feed a model. However, while some models are proven to be superior to others in a given task, in most cases, the data selection step is a crucial aspect of a forecasting system, thus, it cannot be overlooked.

In this section, we will discuss the most common input data strategies, as per the conducted scientific literature analysis.

4.4.1 Data selection

As the famous adage "Garbage In, Garbage Out" implies, we must make sure our data is as relevant as possible. Therefore, we select input variables that are strongly correlated with the variable(s) we are trying to predict.

Several authors studied the correlation between multiple variables with the electric load such as Dedinec and Jovanović et al. with their respective papers entitled "Dedinec [2016]" and "Jovanović et al. [2015] using data from Macedonia and Serbia, respectively.

Both of these papers concluded that exterior temperature is correlated with electricity consumption.

Moreover, a United-States-Environmental-Protection-Agency report showed that a non-negligible proportion of the electricity consumption is dedicated to HVAC (Heating, Ventilation and Air Conditioning), roughly 16% of the USA total electric consumption in 2013, as visible on Figure 4.20

Therefore, we can conclude that the main driver for HVAC use is weather variables such as air temperature and wind speed. Dedinec et al. [2016] also showed that outdoor temperature was strongly correlated with electric load (Macedonia); however they showed negligible correlation with humidity.

On the other hand, Xie et al. [2018] state that the effect of humidity on load forecasting is still not yet understood and requires further research.

Some, such as Ryu et al. [2016], took it further and included not only temperature but also solar radiation, cloud cover and wind speed.

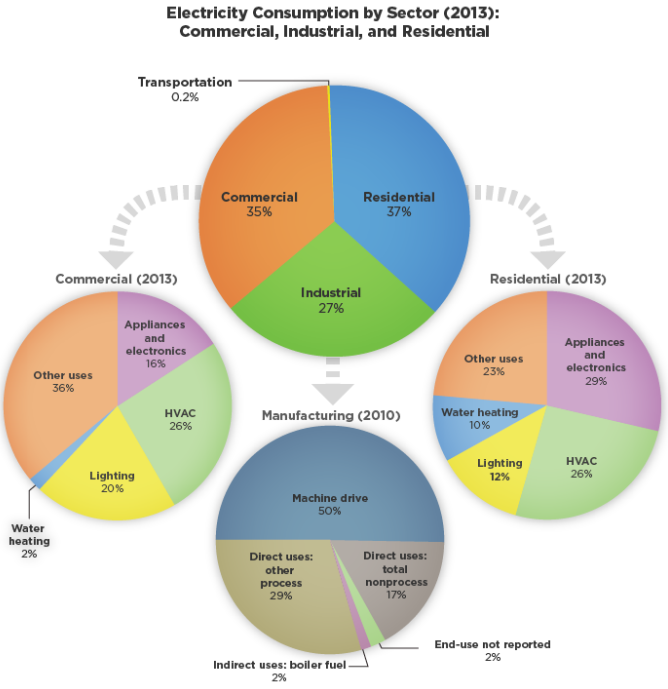


FIGURE 4.20: Electricity use by sector, USA, 2013 (United-States-Environmental-Protection-Agency)

Time related data naturally affects electricity consumption, such as the time of day, the day of the week and month. Sometimes, the day type such as holiday or weekday is also indicated (Ryu et al. [2016]).

Electricity price may also have an influence and is sometimes considered either as a binary flag for the ANN model (cheap tariff due to low demand, Dedinec et al. [2016]) or have its relationship with load modeled separately as part of a larger forecasting system (Ouyang et al. [2017]).

In the context of disaggregated load prediction, Hernández et al. [2014] also included the aggregated forecast load as input. Indeed, forecasts on a single aggregated load are more accurate than the sum of forecasts on multiple values (geographic zones).

When it comes to the time series itself, Dedinec [2016] concluded that the load from the previous day at same hour, the same hour, same day load of the previous week along with the overall consumption the day before all had strong correlations with the load of the next day (forecast).

Marino et al. [2016] explored the impact of time series granularity (1 minute vs 1 hour), concluding that for 60 hours ahead building load forecasting in, the model trained with one-minute resolution showed larger forecasting errors compared to the same model trained with hourly data.

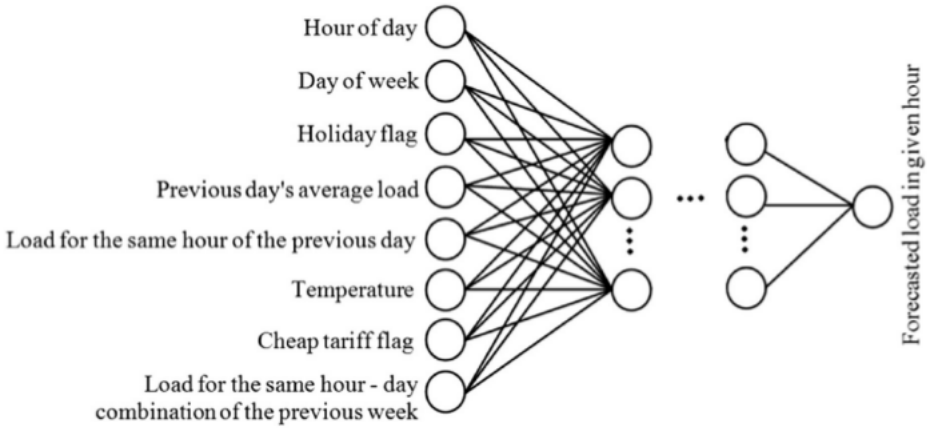


FIGURE 4.21: ANN input values (Dedinec et al. [2016])

In general, input variable selection is still under active research as shown by Xie et al. [2018], Hernández et al. [2012] and Jovanović et al. [2015] recent papers.

However, it should be noted that it is also possible to solely input the X previous values in order to predict the next step. This approach is used by Qiu et al. [2014] as well as Dedinec et al. [2016] where this method is compared with the one described above. In their case, careful selection of input variable led to better results (3.3% vs 3.95% MAPE).

4.4.2 Data preprocessing

When working with machine learning models, it is important to explore our dataset of input variables (selected in the above section). Datasets are very rarely perfect as they often contain missing data and outliers. Usual techniques to detect and solves these issues are applicable in our case too. The most common strategy in handling missing data is to take advantage of the continuous and slowly changing (inertia) nature of the variables of interest. As an example, if a temperature reading is incoherent (outliers) or missing for a given time step, we can replace it with the average of the surrounding values or the previous value (given that the time step is small enough).

More complex methods can also be employed to deal with outliers such as Principal Component Analysis (PCA) as did Hernández et al. [2014].

Normalization is necessary because some models (such as Neural Networks) rely on the distance[1] between values, yet the ranges of input variables can differ drastically, leading to the domination of larger features (e.g. salary) over smaller ones (e.g. age) and leads to poor model performance.

[1] Typically Euclidean distance.

Feature scaling also has an other benefit with neural network as it tends to makes gradient descent converge faster. This is because gradient descent steps are proportional to the learning rate and the feature value. As gradient descent occurs simultaneously over all features and typically using the same learning rate on all of them, a given learning rate cannot be optimal for all features if they are not on the same scale. Re-scaling all features ensure that the gradient descent steps have the same magnitude on all features, hence make convergence faster. Some optimizers such as Adam have per-feature learning rates and may be less sensitive to this issue. However, these optimizer still use a unique learning rate reference and benefit from feature scaling.

This phenomenon is covered in more details by Reif [2017].

More advanced transformation techniques such as using power transforms, for example, Ouyang et al. [2017] used a single-parameter Box-Cox transformation (Box and Cox [1964]) on the load data.

It is important to note that any transformation applied to the input data must be invertible so that the model's output can be converted to concrete physical data.

Additionally, when using categorical variables as input to a machine learning model, it is recommended or even necessary to apply one-hot encoding to these variables. This is so that the model doesn't attempt to model any relationships between categories such as 1 and 2 or "$T1$" and "$T2$". If a variable has N possible categories, $N-1$ binary inputs are feed into the model. Only one of them can be equal to 1 at a time and corresponds to the variable being equal to the N^{th} category.

All previous mentioned processing techniques are rather standard and not problem-dependent, however a few additional data transformations are sometimes performed in the context of electric load forecasting.

Drezga and Rahman [1999] showed that encoding periodic (such as the time of day) variables in the form of sine or cosine functions has a positive impact on ANN performance. As such, this transformation has been applied by Hernández et al. [2014] and Ramezani et al. [2005].

Many other statistical transformations can be applied to time series, one of them was Box-Cox transformation, another is time series differencing. In Kuremoto et al. [2014], it was showed that the latter technique positively affect forecasting performance (Kuremoto et al. [2014]) on both statistical models (e.g. ARIMA) as well as machine learning models (e.g. DBN, MLP). The goal of differencing a time series is to remove or reduce temporal dependence such as trend and seasonality (e.g. week, seasons), in turn reducing the overall time series complexity and stabilizing the mean.

4.5 Evaluation

In order to be able to compare models, we first have to define commonly used statistical metrics. These are the Mean Absolute Error (MAE), Mean Square Error (MSE), Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE) and Hit Rate (HR - Ouyang et al. [2017]).

$$MAE = \sum_{i=1}^{N} \frac{\left|\hat{Y}_i - Y_i\right|}{N} \quad \text{Mean Absolute Error} \tag{4.16}$$

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(Y_i - \hat{Y}_i)^2 \quad \text{Mean Square Error} \tag{4.17}$$

$$RMSE = \sqrt{MSE} \quad \text{Root Mean Square Error} \tag{4.18}$$

$$MAPE = \frac{100}{N}\sum_{i=1}^{N}\left|\frac{Y_i - \hat{Y}_i}{Y_i}\right| \quad \text{Mean Absolute Percentage Error} \tag{4.19}$$

$$HR = \frac{1}{N}\sum_{i=1}^{N} I(i) \quad \text{Hit rate (Ouyang et al. [2017])} \tag{4.20}$$

$$I(i) = \begin{cases} 1 & \text{if } \frac{\hat{Y}_i - Y_i}{\hat{Y}_i} < \text{threshold} \\ 0 & \text{if } \frac{\hat{Y}_i - Y_i}{\hat{Y}_i} \geqslant \text{threshold} \end{cases}$$

For reference, Ouyang et al. [2017] defined the threshold as 0.07.

N : Number of testing samples

$\hat{Y}_i$: Predicted value

Y_i : Actual value

The models developed as part of these project will be compared between each other under a variety of settings. Models tested on NYISO dataset will also be compared to NYISO forecasts.

The tested models will include Deep Learning architectures such as LSTM, CNN and Dense models using data from sources discussed in Section 3 on the task of day-ahead load forecasting.

4.6 Discussion

First, we will compare statistical methods (ARIMA, DSHW) with computation intelligence methods, then compare deep neural networks models with other data-driven methods such as the SVR. Lastly, we will compare the most common deep learning architectures used to solve this problem.

It is important to note that while various architectures typically offer overall different levels of performance, the effect of data preprocessing, and data sources (aggregation level, input variables, geographical location...) can have a very noticeable impact on the performances of these algorithms. Unfortunately, identical datasets are rarely used throughout papers and the preprocessing steps are hardly comparable too. ANN architectures can also be different while still being under the same umbrella name such as 'RNN', making a fair comparison even more complex.

Note that superlatives such as "better", "superior" and "reliable" when comparing models refer to the models' forecasting power in a similar setup, usually measured using the MAPE metric. A lower MAPE reflects smaller forecasting errors. Other aspects such as implementation and computational complexity (training time and inference) are also to be considered but are not the main focus of this literature review.

4.7 Statistical methods compared to artificial intelligence techniques

The forecasting performance of some data-driven architectures has been shown by multiple authors to be superior compared to statistical methods (ARIMA, DSHW...). Ryu et al. [2016] compare several models including ARIMA, DSHW, MLP, and DBN using 7 years of Macedonian daily load profiles. The Double Seasonal Holt Winters (DSHW) model showed good prediction accuracy (2.55% MAPE) for hourly day-ahead forecasting, while the ARIMA model offered the least reliable forecasts with a MAPE of 3.29 %. On the other hand, the MLP model had a MAPE of 2.98 %, worse than the DSHW but better than ARIMA's. They also developed a deep learning architecture (Deep Belief Network) and showed that it was capable of reaching a MAPE as low as 2.19 %. Similarly, Dudek [2015] shows that ARIMA has a 70% higher MAPE when compared to an ANN on Polish load data. It is important to note that these methods often require strong pre-processing sometimes including energy expert knowledge in order to exhibit such levels of performance, while Artificial Neural Networks usually require no domain-specific data transformation.

Overall, complex data-driven models such as DBN and CNN can yield superior performance but simpler ANNs may not be capable to capture the data complexity enough to provide better forecasts when compared to statistical methods.

4.8 Non-ANN based vs ANN-based models

The most commonly used non-ANN data-driven models are the Support Vector Regression (SVR) and the Random Forest (RF) ensemble model. Many variants of the SVR have been developed, oftentimes for the purpose of a single research paper, making comparison without re-implementation difficult.

While multiple authors (Ouyang et al. [2017], Kuo and Huang [2018]) showed that the SVR typically delivers better forecasts than shallow neural networks (MLP), this architecture most often couldn't compete with deep neural networks. However, Qiu et al. [2014] showed that the SVR model is capable to outperform many ANN based models such as MLP and even DBN. It still can be argued that the data complexity remained low with only the last 24h hourly loads being used as inputs. Complex architectures such as DBNs are more suited for complex datasets such as the one used by Ouyang et al. [2017] which includes many additional inputs variables including weather information (e.g. temperature, humidity, pressure, etc) and electricity price. Their work showed that DBN (5.47% MAPE) and even MLP (6.10% MAPE) are capable to outperform the SVR model (6.37% MAPE) on day-ahead and week-ahead load forecasting.

Kuo and Huang [2018] included a Random Forest regression model showing great performance with a MAPE of 10.87% when compared to the MLP's 15.47% and their proposed model's (CNN) 9.77% MAPE. Dudek [2015] showed less impressive results with the Random Forest model having similar performance compared to a shallow neural network. Lahouar and Slama [2015] build a domain expert crafted RF model capable to perform better than an MLP on a Tunisian hourly load dataset. One could argue that as a lot more work and fine-tuning has been poured into their own model, the comparison is slightly biased. However, their work - as most others presented papers in this dissertation - is peer-reviewed. They also compared their model with an SVM and provide their results with a seasonal and day of the week breakdown showing that while the SVM model performs much better than the MLP in Fall (35% lower MAPE), it had 60% higher MAPE in the Winter season.

We can conclude that on the task of load forecasting, the superiority of ANN based method has been proven by multiple authors using a variety of datasets, input variables and models.

4.9 Deep architectures comparisons

The focus of this dissertation is on deep learning architectures, as such, we will explore the performance advantages of deep architectures compared to shallow ones, then compare various deep learning models.

Shi et al. [2017] provide a great insight on deep versus shallow architectures, comparing RNN models using from 1 hidden layer, up to 5 hidden layers on two load datasets. They conclude that the deepest RNN exhibits a 23 % lower MAPE on aggregated load data (New England) and 5% lower MAPE on an Irish dataset consisting of 100 households, when compared to a shallow RNN (1 hidden layer). Both results are in accordance with the work of Ryu et al. [2016] for Korean aggregated data (30-35% MAPE improvement from MLP to DBN) and of Amarasinghe et al. [2017], showing between 2% (CNN) and 10% (LSTM) RMSE improvements on building load forecasting (Canada). Dedinec et al. [2016] show more moderate gains on aggregated data (Macedonia) with a 1.81% MAPE lead for the DBN architecture over the MLP one.

Tong et al. [2017] compared an SVR model with ANN model along with their proposed model consisting of stacked auto-encoders, an architecture similar to Deep Belief Networks. They concluded that the SVR model had lower MAPE compared to the ANN model but about 60% higher overall MAPE (aggregated results for day-ahead forecasting on 3 US load datasets) when compared to their proposed deep architecture.

Hosein and Hosein [2017] confirm other results when comparing load forecast MAPE of multiple models including MLP, SVR, several variants of DBN, DNN, CNN and RNN. They also provide source code for the deep learning architectures at Github 'smhosein' repository (using Keras toolkit). Their work showed that Deep learning architectures offer greatly superior performance to statistical methods and that their DNN model, when trained long enough (400 epochs) is capable to outperform other deep architectures. Their results also show that RNN-based architectures (such as LSTM) require much longer training times (7-30 times more than DBN, 14-60 times more than MLP and up to 1500 times more than linear regression, the fastest model with regards to training time). Their data comes from residential customers (smart-meters) collected over one year (hourly readings) in a tropical region (temperature fairly constant over the months).

Deep learning architectures are currently the state of the art in electric load forecasting but carry a number of challenges (discussed in Section 4.3.3.1).

Kuo and Huang [2018] and Amarasinghe et al. [2017] provide 3 day ahead forecasting metrics for a CNN, an LSTM and other models. Kuo and Huang [2018] showed that the proposed CNN model has lower MAPE compared to MLP, RF, SVR and LSTM as opposed to Amarasinghe et al. [2017]'s CNN model, unable to beat their LSTM reference model (Marino et al. [2016]) but was able to outperform the MLP and SVR models.

CNN and LSTM both exhibit great performance and, as often in the field of Machine Learning, it is not possible to determine in advance which will perform better on a different problem different dataset such as the ones used in this project.

He [2017] proposed some more complex architectures as seen in Figure 4.22, consisting of 2 different networks : a CNN used as a load feature extractor (instead of manually selecting relevant load variables such as previous day-same hour load) while the RNN models the temporal dynamics in the load data. Additional input variables are also used : hourly temperature (minimum and maximum) along with some time information. Other models were included in their studies, a Deep Neural Network (MLP with 3 hidden layers) and a SVR model which can be used as a reference point (1.72% MAPE). The DBN architecture offered improved forecast performance at 1.664% MAPE but their Parallel_CNN_RNN model was the most reliable at 1.405% MAPE. The data used consisted of a 3 years hourly load dataset from a north Chinese city.).

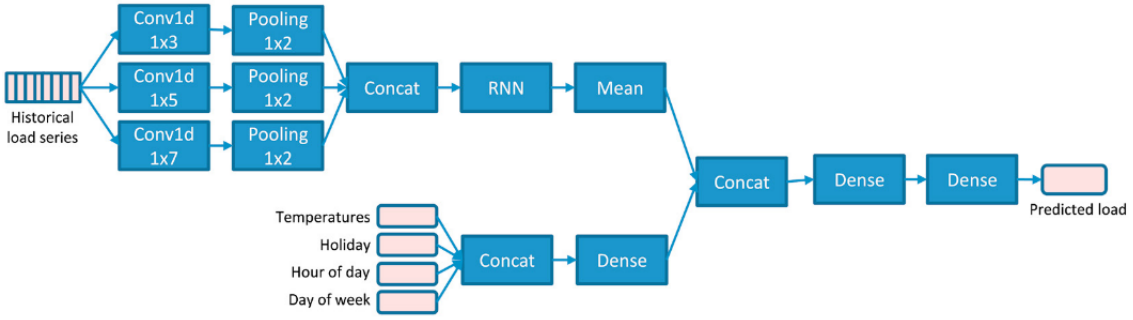


FIGURE 4.22: He [2017] proposed architecture

Other more complex architecture such as ensemble models also offer competitive performance. Qiu et al. [2014] trained an ensemble of 20 DBN on the first 9 months (2013) of Australian data (with 30 minutes granularity) for various epochs (from 100 to 2000). Their output is aggregated using an SVR model and the system yields a 3.69% MAPE compared to 4.20% for a single DBN, 4.58% for the MLP model and 3.85% for the SVR model only. These were tested on the following 3 months of data, which can be criticized since it is known that load profiles have a dependency on the season of the year, however, all models used the same data, keeping the comparison fair.

The proposed architecture is shown in Figure 4.23. Ensemble models have several advantages discussed in Qiu et al. [2014] and can be aggregated in various ways as explained in Hassan et al. [2013]'s work.

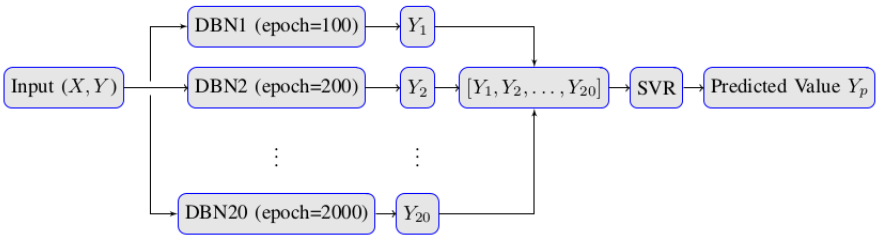


FIGURE 4.23: Qiu et al. [2014] ensemble architecture

Some authors used alternative learning methods such as Particle Swarm Optimization (PSO) in order to either discover optimal weights (in place of gradient descent), optimal ANN structure (in place of trial and error) or to find the optimal learning rate (in place of hand fine-tuning / trial and error). While some showed improved performance over more statistical learning and parameter optimizations methods, these Evolutionary Algorithms (EA) methods add another layer of non-determinism to the system and do represent a non-negligible complexity and uncertainty in the system as a whole.

More complex methods have been shown to yield better prediction at the expense of a usually less straightforward implementation process.

Debnath and Mourshed [2018] did a very extensive review containing near 500 related papers including 50 different models for electricity-related forecasts (load, price, weather variables...) on various geographical scales (region, country, worldwide) and horizons (short, medium, long). The conclusions are that artificial intelligence methods demonstrate better performance than statistical methods. ARIMA and MLP are respectively the most commonly used statistical and AI-related methods. When using ANN, the most commonly used number of hidden layers is 3, with usually (80-90%) less than 10 neurons per hidden layer. Many less commonly used methods, including Fuzzy logic, Evolutionary Algorithms (EA) and various hybrid architectures are also included in this review.

While the most common standard deep learning algorithms for load forecasting have been compared in this literature review, it is important to keep in mind that there is no single best model for every machine learning task ("No free lunch" theorem). Moreover, the source and pre-processing of input data are as important, if not more than the model itself.

Chapter 5

Requirements Analysis

In order to achieve the goals stated previously, in Section 1.1, the following requirements must be met throughout the project development phase.

In this section, we will identify the various requirements of this project :

- Normative requirement
- Personnel involved.
- Theoretical and practical knowledge required.
- The software and hardware necessary to conduct the experiments.
- The additional weather datasets that are not yet available to this day.

5.0.1 Normative requirements

Forecasting capabilities :

Mandatory : The system should be capable to offer forecast more accurate than the persistence model which consists in repeating the previous known output.

Optional : The system shall be able to produce high quality forecasts with typical metrics similar to the state of the art, described in the literature review. In which case, comparison with the persistence model is not necessary.

Features :

Mandatory : The module should be able to handle multiple models and datasets and switch as seamlessly as possible between them.

Optional : The module shall be easily reusable as to allow end users to configure it extensively and use it for purposes that are out of the scope of this project (e.g. time series forecasting in another domain.).

End-user usage :

Mandatory : The system must be capable to generate outputs without the need for programming language knowledge.

Optional : The system shall offer a clear API to be used by developers. Additionally, a graphical user interface allowing users to easily take advantage of the software would be suitable. The system shall also expose a REST API for distant use via the Internet.

Developer usage :

Mandatory : The source code of the system should be readable and as clear as possible to ensure its behavior can be understood by knowledgeable people. This can be done through the use of a carefully crafted software architecture and comments.

Optional : The system shall offer a documentation describing in a single document the usage of the system. The system shall also be easily modifiable by others via the use of a flexible software architecture.

5.0.2 Personnel

This project being a MSc level dissertation, I, Jonathan MEYER, will be the main person conducting the experiments. However, my supervisor, Dr. Katrin Solveig Lohan, will closely monitor and assist when possible, to ensure project goals are achieved. Regular meetings with the industrial partner, Smarter Grid Solutions Ltd. will also be scheduled in order to gather their feedback on the ongoing research and development. These meetings will either happen via online chat applications such as Skype or in person in the company's office, in Glasgow. Transportation will be either by bus or train and the cost covered by the project sponsorship.

An intern at Heriot-Watt University, specialized in energy systems may also bring some additional theoretical knowledge as to build the most reliable forecasting system. In return, practical knowledge, mostly when it comes to programming will be provided by myself.

Moreover, a PHD, currently conducting research at Heriot-Watt University in the field of Machine Learning, has also agreed to offer theoretical and technical support when necessary.

5.0.3 Knowledge

Additional theoretical knowledge

Additional theoretical knowledge on Machine Learning and especially on Deep Learning will have to be gained, mainly through the completion of various renowned online courses including deeplearning.ai and fast.ai lectures on Deep Learning. Note that these lectures are not free of charge but remain accessible without dedicated sponsorships.

Practical knowledge

While the literature review of this dissertation helped in gathering theoretical information on the problem of load forecasting, practical knowledge needs to be gained.

The used software suite was very rarely mentioned in the literature, however, only a few Deep Learning toolkits are currently commonly used such as TensorFlow (Python), Caffe (C++) and PyTorch (Python).

Keras, a high-level neural network library will be used. Keras can interface with both Tensorflow - of which it is the official high-level interface - Theano and CNTK. Both Keras and Tensorflow are widely used and with strong community support. The choice of Python over C++ was made to make development faster.

In order to gain practical knowledge using Keras, tutorials from a large collection gathered by the creator of the Keras toolkit (François Chollet) will be explored. When it comes to performing data preprocessing, the pandas Python library will be used.

5.0.4 Hardware

Deep Learning applications are known to be very computationally expensive and many models greatly benefit from the use of Graphical Processing Units (GPUs). My personal laptop is fitted with a mid-range modern GPU (Nvidia GeForce GTX 1060), with Tensorflow and Keras configured to take advantage of its parallel processing capabilities.

Additionally, the Robotarium compute cluster, managed by the Heriot-Watt University can be used if necessary. It features high-end hardware including GPUs which can be leveraged using the aforementioned toolkits.

5.0.5 Weather datasets

Considering the impact of weather variables, it may be beneficial to obtain historical along with forecast weather data from external sources. Such data doesn't appear to be widely available, especially historical data.

Possible weather dataset sources

The ideal data source has the following characteristics :

- Time : Historical and forecast data (at least day-ahead).
- Space : Multiple stations per region (e.g UK, NYISO zones)
- Variables : Hourly temperature measurements, or better. Other weather variables such as wind speed and humidity may also be worth exploring if available.
- Cost : Free of charge.
- Access : API or bulk data download.

Many paid services would match these constraints, such as OpenWeatherMap Historic Bulk API and The Weather Company. Note that Wunderground also provides an API using The Weather Company's data.

Free of charge sources include METAR archives and data from governemental organizations. METAR archives are collections of weather report mostly used at airports and typically contain wind speed and direction, visibility, cloud coverage, temperature and pressure, on a 30 minutes basis.

Governmental data, such as NOAA[1] and Met Office also give access to some open-source data in a less standardized format.

After a relatively long period of research, a suitable source is finally identified : NOAA's NCDC[2] Integrated Surface Dataset. This data is discussed in Section 3.2.

[1] US National Oceanic and Atmospheric Administration.
[2] US National Centers for Environmental Information

Chapter 6

Professional, Legal, Ethical, and Social Issues

6.1 Professional issues

As a collaborative project between 3 parties, professional conduct from all participating members is mandatory. Also, to guarantee that objectives are achieved in compliance with the project proposal, frequent communication, either in person or through video calls is necessary.

The code is developed following the Official Style Guide for Python code and commented to ease the reading, validation and further development.

6.2 Legal issues

All information and software is obtained legally, directly from the associated sources and free of charge. All data sources used in this project are anonymous (aggregated) and accessible for all on the Internet. The Term of Service and other legal disclaimers will be consulted before performing any work with said data sources and pieces of software.

Source code ownership - written as part of this project - is discussed with the stakeholders, namely the Heriot-Watt University, Smarter Grid Solution Ltd., Dr. Katrin Solveig Lohan and myself. Legal advice from a lawyer has been queried to ensure proper decisions are made. The source code is considered Open Source Software, delivered under the MIT licence, included in the public GitHub repository where the source code will soon reside.

6.3 Ethical aspect

As no personal data is to be used in this project, and the aim of this research being to improve a public service (electricity generation and distribution), it is not believed that there exists any adverse ethical aspects, given that the software is used as intended.

However, this project can be used as a pseudo-generic[1] time series forecasting tool, as such could be applied to many domains, some of which may have a significant ethical impact.

6.4 Social aspect

As discussed in the "Project Description" Section of this dissertation, load forecasting has strong financial impacts. The industrial partner estimates a potential turnover increase upwards of 250000£ in the first year only after the deployment of the developed forecasting module.

Along with that, an approximated 17000 Tonnes of CO^2 are expected to be saved every year and potential new jobs are likely to be created as a result of this project.

All these consequences affect the wider public with lower electricity prices, lower pollution levels and more job offerings.

[1] Given some potential code modifications

Chapter 7

Module description and implementation

7.1 Module description

7.1.1 Goals

As described in the "Aims and Objectives" section of this report, the goal of this module is to be able to forecast the electricity load / demand. The literature review also revealed that the weather has a strong impact on the load, as such the module should be capable to handle weather information. A selection of machine learning models are compared, along with a variety of different settings using data from 3 sources as described in Section 3.

7.1.2 Technologies

This module is built using the following software tools:

- Xubuntu 17.10
- Python 3.6.3
- Python libraries
  - Tensorflow 1.9.0 (tensorflow-gpu)[1]
  - Keras 2.2.0
  - sklearn 0.19.1

[1] I highly recommend to use this specific version as Tensorflow 1.8 crashes on an instruction used to prevent memory overflow.

  - pandas 0.23.3
  - NumPy 1.14.5
  - matplotlib 2.2.2
  - seaborn 0.8.1
  - TinyDB 3.9.0.post1
  - json 2.0.9
  - GeoPandas 0.3.0
  - Shapely 1.6.4.post1
  - LiveLossPlot at Github as of 29/07/2018.
  - Reverse geocoder 1.5.1 at .

- GPU support
  - NVIDIA GTX 1060 Max-Q GPU
  - NVIDIA driver 387.34
  - NVIDIA CUDA 8.0.61
  - NVIDIA cuDNN 6.0

- Jupyter notebook 4.3.0

7.1.3 Processing pipeline

1. Target data processing
   (a) Load historical data.
   (b) Repair errors in the data (e.g. missing values, duplicates, outliers...).
2. Support dataset : Weather data processing
   (a) Load the corresponding[1] historical weather data[2]
   (b) Filter weather data (e.g. by date, location, data quality...)
   (c) Combine weather stations based on their location[3] as to reduce dimensionality[4] and increase data quality.[5]
   (d) Drop clusters with too many errors.
   (e) Repair remaining errors (e.g. missing values, duplicates, outliers...).

[1] Same location
[2] One time series per weather station, per weather variable (e.g. Air temperature).
[3] Geo-clustering : Grouping of GPS coordinates (stations) based on their distance to one another.
[4] Have fewer input variables, in turn reducing computational needs when training.
[5] By combining similar times series containing abnormal values, we obtain one with fewer of them.

3. Machine learning pre-processing
   (a) Combine target and support data.
   (b) Encode time in a suitable format.
   (c) Forward-propagate historical target values.
   (d) Transform features to a suitable format for ML models.
   (e) Split data into cross-validation sets.
4. Machine learning training
   (a) Initialize a machine learning model (e.g. 2 hidden layers deep LSTM)
   (b) Train a model instance on the training set of each cross-validation set.
5. Machine learning evaluation
   (a) Evaluate each model[1] on their respective training and testing data.
   (b) (Optional) Plot results and give various metrics (i.e. global MAPE, hour-based RMSE ...)
   (c) (Optional) Store results into a NoSQL database for later analysis.

7.1.3.1 Target data processing pipeline

The target data is loaded, internally transformed into a standard format then passes through a pre-processor designed to detect and correct possible errors.

A variety of 'detectors' and correction strategies are implemented such as :

Detectors

- Zeros detector : Detects zero values.[2]
- Extremes detector : Detects values exceeding some thresholds based on data percentile information.[3]
- Extreme hourly detector : Extreme detector using hourly based percentile values.
- Step detector : Detects samples having a two-way[4] absolute percentage difference above a threshold based on global percentile values. This detector can be used to detect abrupt changes in the data.

Examples for each detectors are visible on Figure 7.1. Note that some detection overlap exists, for example the extreme detector can also detect zero values in most cases.

[1] One per cross-validation set.
[2] Should only be used with time series not containing any legitimate 0 values.
[3] For example : Values 1.25x larger than the overall 99 percentile are considered abnormal.
[4] With reference to its preceding and following sample.

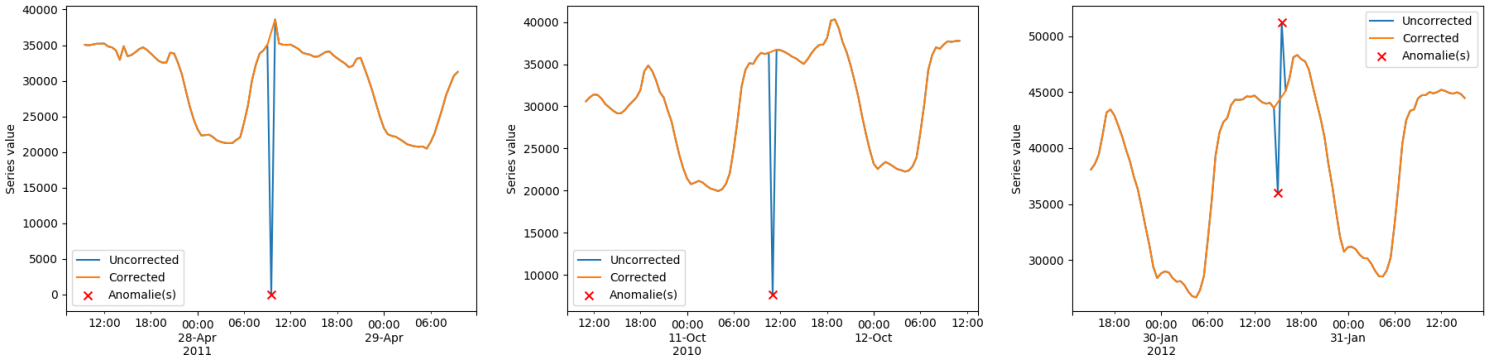


FIGURE 7.1: Zero / Extreme / Step detectors examples

Note on the extreme hourly detector : While functional, by its nature[1], it tends to generate false positives, especially on high variance datasets.

Correction strategies

- Interpolation : Linear interpolation between data points.[2]
- Replace by NaN : Used when external means of corrections are to be used.

More advanced corrections strategies including data modelling (e.g. ARIMA) can be used as described in Zhang et al. [2017].

However, considering the good quality of the experimentation datasets (as seen in Section 3), the implementation of such methods within the scope and duration of this project was deemed unnecessary.

All errors and their suggested corrections can be visualized as in Figures 7.1 by setting the debugging flag "MANUAL_ANOMALY_VALIDATION" to "True" in the "Preprocessor.py" file.

7.1.3.2 Weather data processing pipeline

NCDC[3] hosts multiple datasets containing data for many weather stations across the globe which are managed by the United States Air Force (USAF).

[1] Expecting very cyclical data.
[2] A linear interpolation is suitable because the series' time step is constant.
[3] National Climatic Data Center.

a) Load the corresponding historical weather

The loading process is straightforward but has been optimized due to the large quantity of data. Optimization mainly comes from the early column selection[1] as well as tight variable typing (i.e. limiting floating point precision when relevant, storing data as numerical values instead of string when possible etc...).

Weather loading and processing performance optimization using UK data :

	RAM	Total weather data processing time
Before	875 MB	1 min 33 sec
After	255 MB	1 min 20 sec

TABLE 7.1: Weather processing pipeline optimization metrics

b) Filter weather data

Then, data is filtered to match the desired date span (e.g. 2008-2010 only, mostly for testing purposes) and locationStations with poor data availability (e.g. some stations have only a few datapoints...) are also removed along with duplicated ones.

c) Combine weather stations

Motivation :

722 stations on the UK territory (including outside mainland) are available, of which 180 have data within the period of interest - defined by the load dataset containing measurements from April, 1^{st} 2005 to March, 16^{th} 2018 (date of data collection).

When it comes to the New York state, data from 126 stations is provided, of which only 20 cover the entirety of the NYISO load dataset[2].

While 20 additional input features, per weather variables (e.g. air temperature) would have a measurable but limited impact on pre-processing and training time, adding 180, 360 (e.g. air temperature and wind speed) or even 480 input features would lead to performance issues. Additionally, these features would be highly correlated with one another (due to geographic proximity) and the value of each features would be small.

Solution :

The suggested approach is to select a subset of stations that best represent the overall population of stations.

More explicitly, this is a geo-clustering problem.

[1] Only the relevant columns are loaded. More often than not, all are loaded then unnecessary columns removed.

[2] May, 26^{st} 2001 to May, 26^{th} 2018 (date of data collection)

A multitude of clustering algorithm exist, the most commonly used being K-means. Internally, K-means uses the euclidean which implies that our points shall be laid out on a plane. However, the Earth resembles a sphere, as such, the great-circle distance, also known as Haversine distance gives better distance approximations, especially for points that are far from the equator (where the curvature of the Earth becomes significant).

An alternative is the DBSCAN algorithm[1], a clustering algorithm often recommended for spatial latitude-longitude data.

Another algorithm, called Mean Shift (Fukunaga and Hostetler [1975]) yields more satisfying results despite rarely being suggested

A clustering example, generated using the software developed as part of this thesis, is visible on Figures 7.2. We can see a large decrease in the number of GPS coordinates, yet the space is covered quite uniformly - with more stations in originally more densely population areas (weather stations).

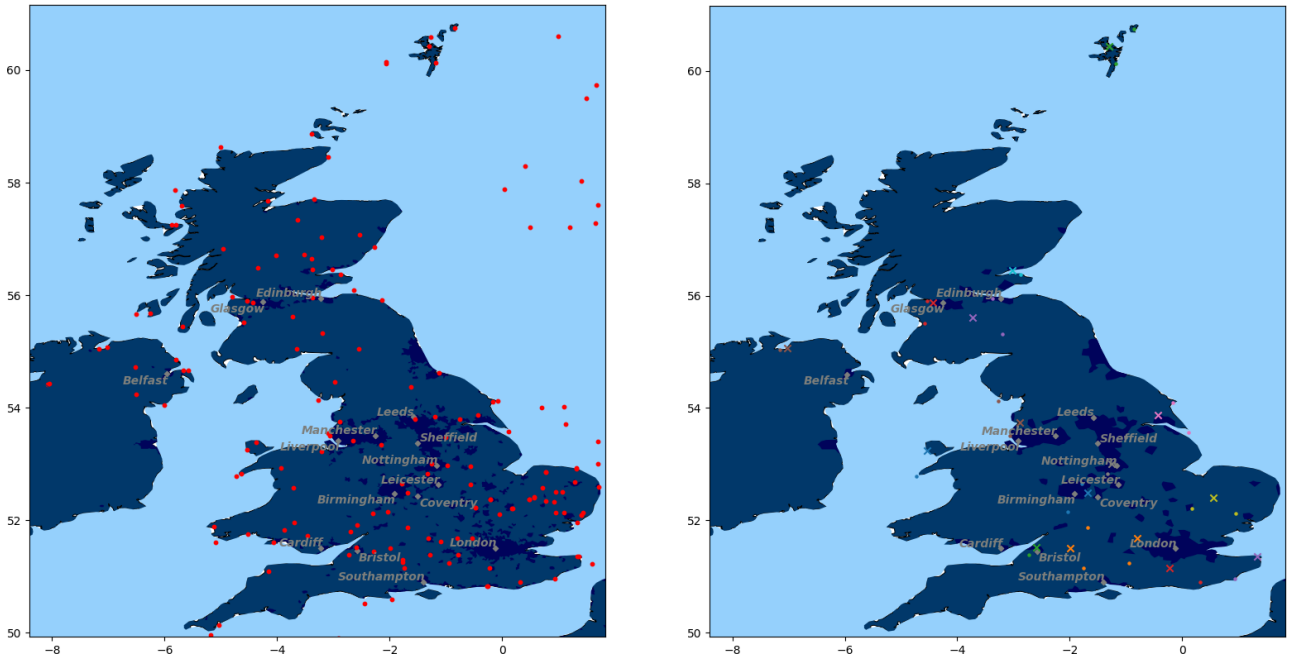


FIGURE 7.2: UK weather stations : Before / after filtering and clustering

Legend :

- Red dot : Weather station.
- Colored cross : Cluster's centroid weather station (unused)
- Colored dot : Weather station from a given cluster.

Note that due to the limited number of colors used, some clusters may have the same color, yet are not related in any way.

[1]DBSCAN : Density-Based Spatial Clustering of Applications with Noise. See Ester et al. [1996]

Contrary to K-means, both DBSCAN and Mean Shift algorithms cannot directly take a number of desired cluster as parameter and instead take a parameter called 'eps' and 'bandwidth' respectively. However, the module API provide such a parameter and try to circumvent this issue by doing a linear search over the possible value for those parameters[1] and find the one yielding the desired[2] number of clusters (of a given minimum size).

An additional pre-clustering step to further improve data quality is recommended. Indeed, some stations are located far from the load / demand regions, for example in the oceans, on remote islands or even outside the UK territory (e.g. Antarctica).

Some of these outliers are detected automatically through the use of a reverse geocoder[3] and a set of constraints on the various retrieved pieces of information. (e.g. Country code, administration levels [4]...). However, some are not and for best results these are removed before hand.

Once the weather stations are clustered, the times series of each stations within a cluster are combined[5] This greatly increase data quality, filling in many of the missing values.

The next step is to impose the same time series frequency as the target series, indeed, some weather stations report measurements every 12 minutes, while some do every 2 hours. Some even have non-fixed reporting frequency (i.e. 7 minutes interval, then 5 minutes etc).

This step is done via time-dependant interpolation to take into account the fact that some measurements are closer (e.g. 10:57) to a target time stamp (e.g. 11:00 am) than others (e.g. 10:42).

d) Drop clusters with too many errors

Additional filtering is done where clusters with unsatisfactory data quality - defined by the number of missing records within a given time span (2 hours by default) - are dropped.

e) Repair remaining errors

The remaining missing values are replaced by the hourly average on the same month (and same year).

[1] Range estimated experimentally using the UK weather dataset.
[2] As close as possible
[3] Provided by Python offline reverse-geocoder
[4] See UK administrative geography for more information.
[5] By taking the mean and ignoring missing values.

Conclusion

This long and complex pipeline was built to be flexible, robust and yield the highest possible quality data even when given sparse data containing abnormal values.

The need for such process was not foreseen during the project planning phase and yet required a significant amount of time. However, it highlights the importance of data pre-processing and reminds that modelling is only a part - typically the smaller one - of Machine Learning.

7.1.3.3 Machine learning pre-processing pipeline

a) Combine target and support datasets

Both input datasets (i.e. load and weather) are combined into a single one. Independent (i.e. features / input variables) and dependant (i.e. target / output variable) are identified. At this stage the dataset looks like this[1,2] :

	TS	Air_Temp_0	Air_Temp_1	Air_Temp_2	Air_Temp_3	Air_Temp_4	W_Spd_0	W_Spd_1	W_Spd_2	W_Spd_3	W_Spd_4
Period											
2005-04-01 00:00:00	29566	8.027272	7.300000	6.6	8.86	6.45	2.600000	4.185714	7.700000	4.00	12.050000
...	...	...	...	...	...	...	...	...	...	...	...
2018-03-16 23:30:00	24512	3.777778	3.416667	4.0	5.92	1.00	6.504444	9.883333	5.466667	8.36	10.133333

FIGURE 7.3: Dataframe after weather and target variable (TS : load) merge.

b) Transform time data format

During the pre-processing steps, the time / date was contained into a single DateTime column[3]. However, most machine learning algorithm, including all neural networks, work with numerical data only. This step is known as encoding. We consider our DateTime object to contain the following components : Year, Month, Day, Hour, Minute. Note that derived features such as day of the week and season can also be used.

[1] Reduced weather station count for easier visualization.
[2] 'TS' is the target variable, in this case, the load.
[3] More specifically a DatetimeIndex, since time is used as an index.

Several strategies exist :

- **'Numerical' encoding :**

  The most straightforward encoding, simply splits the different fields of the DateTime object into different columns, of integer type, which is suitable for ML models. However this methods encodes time in a non-continuous manner. For example, the end (31) of a month is far from the beginning (1) of the following one while these 2 dates are actually just one day apart.

- **Continuous encoding :**

  We can encode all date information as a single continuous variable by using the Unix epoch time.[1]. While this encoding method solves the above problem, it has a major flaw : testing data have a date value that the network never saw before, which limits the generalization capabilities[2].

- **Categorical encoding :** Categorical encoding is typically done through a procedure called One-Hot-Encoding (OHE). This methods has already been presented in the literature review. However this method doesn't take advantage of the cyclical nature of time (i.e. 01/02 comes right after 31/01). This method also generates many input features, slowing down training.

- **Cyclical encoding :**

  Variables (e.g. day in month) are encoded using trigonometric functions : sine and cosine. However, encoding using only one of them doesn't create a bijective function because sine and cosine over 2 Π have 2 possible original x values (e.g. noon and midnight) for an encoded y value. To solve that, we encode using both sine and cosine. Note that the year is not cyclical variable, hence is left as is (numerical value).

At this stage the dataset contains the previous columns along with additional time data (using cyclical encoding):

	Year	sin_hour	cos_hour	sin_day_in_month	cos_day_in_month	sin_day_in_week	cos_day_in_week	sin_day_in_year	cos_day_in_year	sin_month	cos_month
Period											
2005-04-15 00:00:00	2005	0.000000	1.000000	5.665539e-16	-1.000000	-0.433884	-0.900969	0.972118	-0.234491	0.866025	-5.000000e-01
...	...	...	...	...	...	...	...	...	...	...	...
2018-03-16 23:30:00	2018	-0.130526	0.991445	-1.011683e-01	-0.994869	-0.433884	-0.900969	0.961130	0.276097	1.000000	6.123234e-17

FIGURE 7.4: Time encoding dataframe.

[1] Number of seconds since 00:00:00 (UTC), Thursday, 1 January 1970.
[2] Capability to apply what the model learned to unseen data.

c) Forward-propagate historical target values

In order to improve forecasting performance, especially on non-RNN models, we can provide samples containing previous target values. For example, we can store the last 24H of load in each sample, which would give a reference point to the model when trying to forecast. This strategy especially works with short term forecasting due to the strong correlation between the latest target values and the current one.

As such, the module implement a 'forward-propagation' mechanism where it is possible to add to every sample, some previous target values and / or their average over a day. Note that adding previous data for which we would not have the true value makes the prediction process more complex. For example, adding the previous hour load to every sample when doing day ahead forecasting means that this data won't be available prior to training (except for the first time step H+1) and must be replaced by a forecast value.

Note that if historical data with "distance" shorter than the forecasting horizon (e.g. H-4 with day ahead forecasting), models are essentially cheating, using true values in place of forecast values. In a production environment, this would simply results in incomplete datasets fed to the network. While this problem was avoided - mostly because Dense models were not explored due to their poor performance - a "recurrent forecasting" method is implemented in the module and allows training and forecasting in this scenario.

At this stage the dataset contains the previous columns along with historical target data columns :

	Previous_1	Previous_2	Previous_3	Previous_4	...	Previous_12	Previous_336	Avg_1	Avg_7
Period									
2005-04-15 00:00:00	30297.0	32226.0	34585.0	36900.0	...	40033.0	29341.0	37147.833333	37793.083333
...	...	...	...	...	...	...	...	...	...
2018-03-16 23:30:00	25546.0	27134.0	28773.0	30412.0	...	35154.0	25898.0	31453.062500	33372.437500

FIGURE 7.5: Historical target data forward propagation dataframe.

d) Feature scaling

Feature scaling, also called standardization or data normalization[1] is the process of rescaling variables so that they all lie within a given range (typically [-1;1] or [0:1]). This process is presented in the "Data Preprocessing" Section (4.4.2).

f) Split data into cross-validation sets

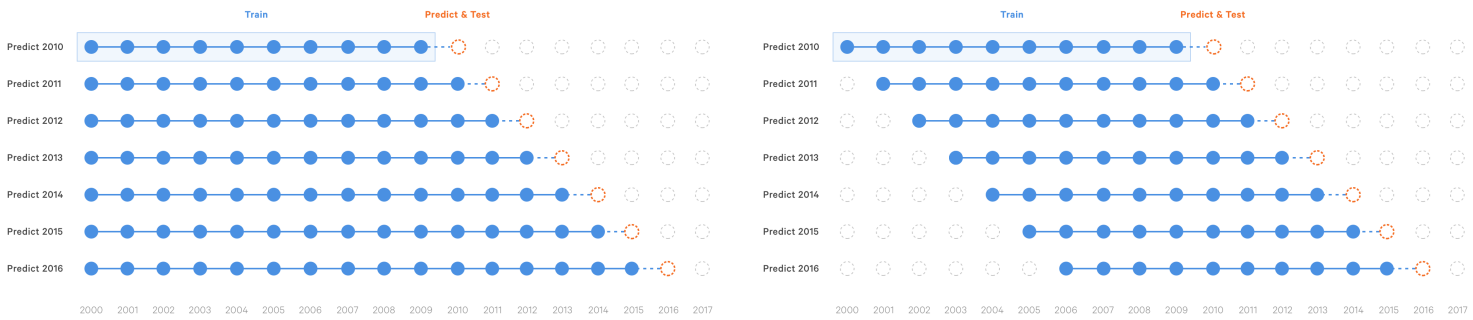
Cross-validation is the process of splitting a given dataset into sub-dataset in order to better evaluate a model's generalization capabilities by training and evaluating it on different sets of data.

[1] Which are actually two of the ways we can apply feature scaling - albeit the most common ones.

Typically, this split is done at random intervals and after shuffling. However, when working with time data, data cannot be shuffled due to the intrinsic ordering. Additionally, I would argue that the splits should only occur at the end of a year to ensure that every day of the year are part of both training and testing sets. Otherwise, a training or testing set could contain only half a year, resulting in a poor model or biased evaluation.

Therefore, both cross-validation and train test splits strategies have been customized to take this into account, only generating datasets containing whole year data, regardless of the numbers of available years of data, number of cross-validations sets and train/test splits.

Given those restrictions, two methods exist : the first one is called 'rolling prediction origin forecasting' and the other 'rolling window forecasting'. The concepts are simple and depicted in Figure 7.6.



(A) Rolling prediction origin (B) Rolling window

FIGURE 7.6: Cross-validation time series forecast methods. From .

In this project, the second method (7.6b) is used as it better tests a model's generalization capabilities compared to the other method which always gives access to a given part of the data (i.e. the first year(s)).

7.1.3.4 Machine learning training

Models

Keras supports a variety of layer types which, when combined, allows us to build LSTM, CNN and feed-forwards 'Dense' models. Deep Belief Networks (DBN) are not officially supported as they are deemed unpopular by the Keras community.

Consequently, 2 types of RNN, CNNs and Dense pre-defined models are available in the module.

Note that the module accepts any custom Keras model, as such, one could use any another model while taking advantage of the pre-processing and evaluation pipelines.

Keras callbacks

Keras offers a callback system, methods that are automatically called during training. Some are already implemented such as the EarlyStopping and ReduceLROnPlateau callbacks.

The point of the first one has already been described while the second one is quite self explanatory : when the validation loss doesn't improve (plateau) for a given number of epochs, the learning rate is decreased. Note that some optimizers (such as Adam) already implements learning rate decay, as such, this feature is not typically used with these optimizers (yet it should not have measurable adverse effects).

The EarlyStopping mechanim requires an additional callback to work properly, tasked with the storage of the latest best model : this is done using the ModelCheckpoint callback.

Other callbacks are also used, such as the PlotLossesKeras[1] used to plot the training and validation losses during training in a Jupyter notebook.

A Tensorboard callback can also be used[2] and allows more complex online visualizations using Tensorflow's visualization toolkit Tensorboard. Some training metrics are also saved on disk for later analysis.

Lastly, a custom callback was created to collect the actual number of training epoch (which can differ from the input epoch count due to the EarlyStopping callback) and overall training time for comparing models.

Learning rate estimation

Additionally, the learning rate was identified using the method described in Smith [2017] [3] with the help of the library at Keras Learning Rate finder.

[1] Non-official, available on Github
[2] As a module flag, active by default.
[3] And vulgarized at Optimal learning rate estimation.

7.1.3.5 Machine learning model evaluation

Introduction

The evaluation process is typically straightforward : Compare your model's prediction $\hat{y}$ against a ground truth y using various metrics, in this case : Mean Absolute Percentage Error (MAPE) and Root Mean Squared Error (RMSE).

Metrics and plotting

As stated previously, models are assessed using the MAPE and RMSE metrics to other developed models but also against the NYISO load forecasts.

Various plotting methods help in gaining insight regarding the forecasting capabilities of the models.

Results storage

A NoSQL serverless database - TinyDB - is used to store results including but not limited to : Testing and training metrics, inference time, data pre-processing settings, machine learning settings (e.g. model, batch size, backend..) as well as, optionally, the detailed results for later full resolution plotting.

7.1.4 Other module functionalities

7.1.4.1 Weather mapper

As seen on 7.2, some geography mapping functionality is available.

All map data was downloaded from NaturalEarth. The files used are "Land (10m)", "admin_0_boundary (10m)", "urban_areas (50m)", "populated_places (10m)" and "oceans (50m)".

7.1.4.2 Benchmarker

A benchmarking utility[1] is available. It allows the reliable comparison of multiple models and settings. Results visualization methods are also available and were used to generate the graphs visible in the "Experimentation results" section.

[1] As a Jupyter notebook : scientific web-based notebook. See Jupyter notebook

7.1.4.3 Dataset builder

For consistency and speed reasons, a utility[1] to build datasets is available. These datasets, along with the settings with which they have been built are stored on disk and can be loaded quickly at a later time.

7.1.4.4 Model builder

Some utility methods to easily build a model (RNN, CNN, Dense) are available. These methods only expect a few parameters, expressed using simple types such as strings and integers. These methods can be used by the user to easily test a range of architectures and are used by the benchmarking utility.

Chapter 8

Experimentation results

8.1 Experimentation results

In this section we will explore the forecasting capabilities of the developed module. Multiple models will be trained and tested on day-ahead and week-ahead forecasting.

However, a machine learning pipeline is not limited to the training of a model but also contains many pre-processing steps that can have a significant impact on the performance (both speed and accuracy).

First, these variables will be introduced and their effect analyzed, then, various models architectures will be compared and lastly the best performing combination presented.

Otherwise mentioned, statistical significance is measured using the unequal variance (Welch) t test.

> "If you want to compare the central tendency of 2 populations based on samples of unrelated data, then the unequal variance t-test should always be used in preference to the Student's t-test or Mann–Whitney U test."
>
> — Ruxton [2006]

The model used for testing the effect of these variables is a 2 (hidden) layers LSTM neural network with 112 units per layer. 112 units corresponds to twice the number of input variables. Each test is run 3 times, or more if results are not statistically significant ($p \sup 0.05$).

The following noteworthy settings are used for testing :

- Maximum epochs : 200
- Early stopping : Active
- Time encoding : Cyclical
- Data from 2008 to 2017, with train test split of (6,2) = 3 cross-validation sets.
- All other settings are set to their best value, or close to.[1]

Note that bar and box plot are ordered by testing set performance (MAPE), from left (smallest error) to right.

If it is believed that some variables may behave differently depending on the model type, experimentation on additional models will be conducted. Due to the slow training speed and lower accuracy, feed-forward dense models won't be explored in much details.

For reference, the NYISO model (unknown) offers day-ahead forecasts with a 3.25% average MAPE for the NYC dataset and 16% (sixteen) MAPE on the MHK dataset. This much higher forecasting error confirms the idea that lower load datasets are more difficult to forecast. We could also assume that NYISO doesn't prioritize forecasting accuracy on this region due to its low load.

8.1.1 Explored variables

A large selection of variables will be explored, as these are expected to have a measurable impact on performance.

8.1.1.1 Electricity data

Error Correction

Given that the data is of good quality, the effects of error correction are non-measurable. However, it is still important to keep in mind its that a dataset containing many abnormal can have a significant negative impact on forecasting performance. A basic replacement of missing values (sometimes encoded as 0, or 999) by the series mean or better yet the neighbouring mean is advisable. Depending on the data, more advanced error detection and replacement strategies should be considered.

[1] Module defaults can be seen in the file named "Load_forecasting.py". These default may be changed when building the datasets : see the "generate_datasets.ipyn" Jupyter notebook.

Forward target propagation

The goal of the "forward target propagation" mechanism is to give each sample more context, beside the date information.

As visible on Figure 8.1, the improvements are clearly visible, with the testing performance improving from a $\sim 5.8\%$ MAPE down to a0 $\sim 3.6\%$ MAPE.

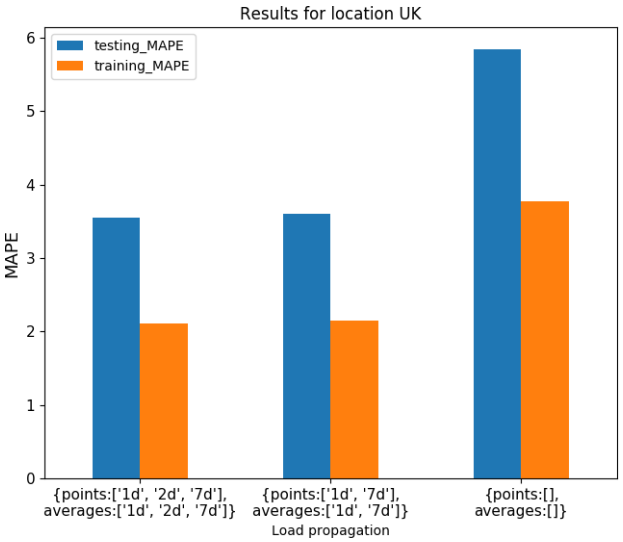


FIGURE 8.1: Forward target propagation effect on forecasting performance

Note that '1d' means that each sample contains the (true) load from 24 hours ago.

Adding even more historical information (leftmost bars) doesn't improve performance in a statistically significant manner (p=0.29). Training times have large variance (due to early stopping and non-exclusive resources), thus cannot be the deciding factor. However, as a rule of thumb, having less features leads to faster training time.

Note that using data from the previous day is only valid when doing day-ahead forecasting, if doing week-ahead forecasting, the true value cannot be used / doesn't exist yet and must be replaced by a forecast value via the use of the "recurrent_predict" mechanism detailed in the "Implementation" Section.

Best performance settings

Forward target propagation : D-1, D-7 and average of D-1 and D-7.

8.1.1.2 Weather data

Here we investigate the effect of weather data for load forecasting. The litterature review showed that it had a measurable impact, and the following experiments show similar results.

Note that the weather data for the New York state was not split to fit the NYISO zones. This wasn't done because detecting which zone the stations fall in isn't straight forward (NYISO zones are not geographical zones) and more importantly because this would reduce the number of weather stations to only a few per zone which even combined could still contain many missing values.

In addition to exploring the effect of using weather data versus not using any, we also explore the effect of each weather variable independently as well as the number of weather clusters.

Indeed, some weather variables may be more relevant than other, meaning that they are more correlated with the target variable.

"'W_Spd':14" means that the wind speed information, clustered in 14 clusters is used as input features. Note that all stations are not clustered and some may not be represented when using only a few clusters. 'RHx' denotes the humidity data.

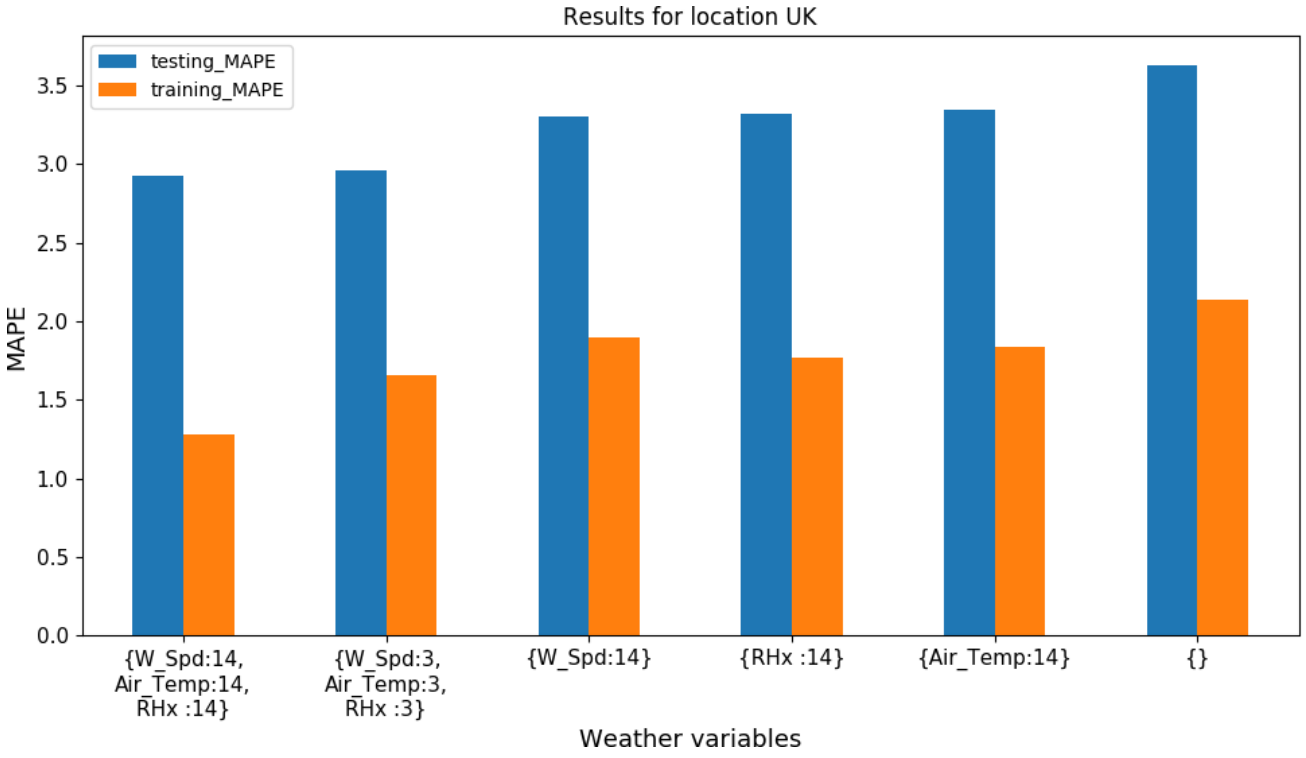


FIGURE 8.2: Weather data effect on forecasting performance using the UK dataset

While each weather variable independently decreases the MAPE, using all of them results in the best performance uplift. The best case scenario (using all weather variables and high cluster count) shows an improvement in forecasting accuracy from ∼3.63% to ∼2.91% MAPE on the UK dataset.

Using only a limited number of clusters (3 per variable) seems to yield lower gains on average on the testing set : ∼2.96% vs ∼2.91% MAPE ($p \approx 0.06$ after 12 runs). However training performance is significantly improved when using more clusters.

The effects on the NYISO datasets (NYC and MHK) vary, possibly due to the nature of the consumers in those areas (residential vs industrial) - residential areas showing a greater sensitivity to weather (air conditioning).

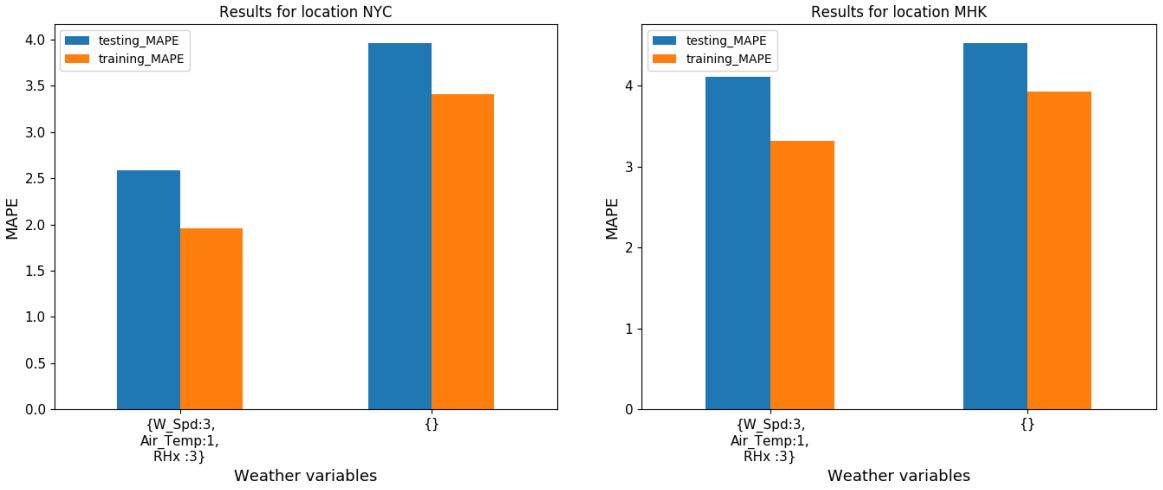


FIGURE 8.3: Weather data effect on forecasting performance for NYC and MHK datasets

A very noticeable improvement (65% MAPE reduction) is experienced when forecasting for the NYC load dataset. The effects when using the MHK dataset are less visible but still positive (∼4.10% vs ∼4.5% MAPE - p=0.00004).

Best performance settings

Weather data presence : Use all weather information.

8.1.1.3 Data encoding

Data normalization

As discussed previously, data normalization is a common data transformation technique in machine learning which aims at suppressing any invalid scaling between variables. Many standardization methods exist and are compared in the following figures.

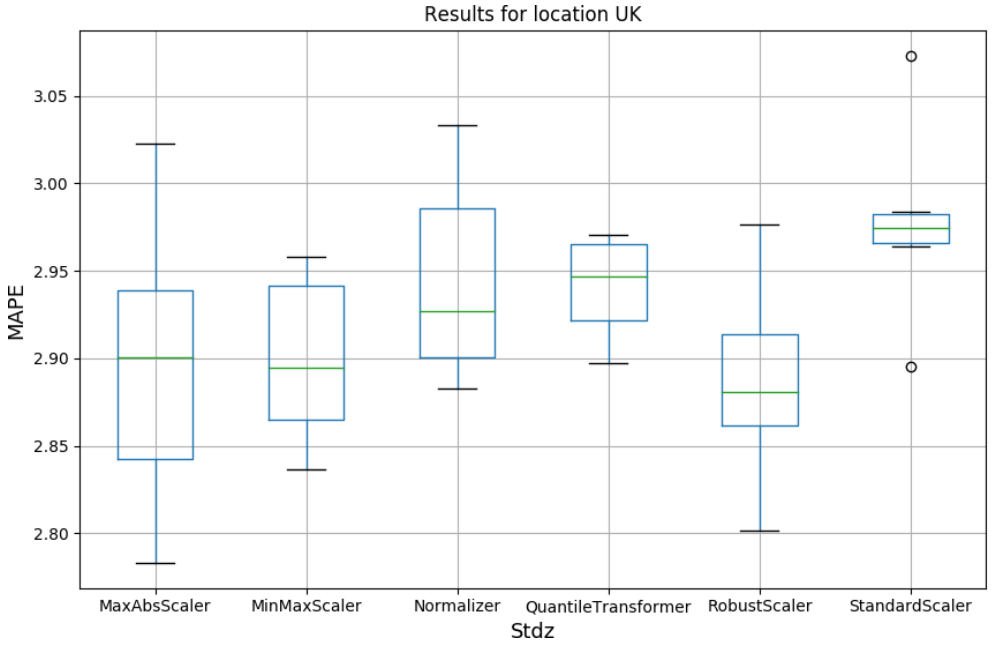


FIGURE 8.4: Data standardization effect on forecasting performance

Despite having 6 runs, no standardizer is statistically significantly better than the others. (one-way ANOVA $p \approx 0.085 > 0.05$; t-test between best and worse mean : $p \approx 0.5 >> 0.05$). Running more times may declare a winner, but only by a very small margin, so small that this superiority may become untrue when using a slightly different dataset or model, as such there is no point in exploring further.

Training on non-standardized data was not attempted but many scientific resources highlight the significant performance difference (both in accuracy and training time) compared to standardized data.

Best performance settings

Data normalization : Use standardization, regardless of the exact formulation.

Time encoding

The effect of time encoding (either categorical or cyclical) are explored in this experiment.

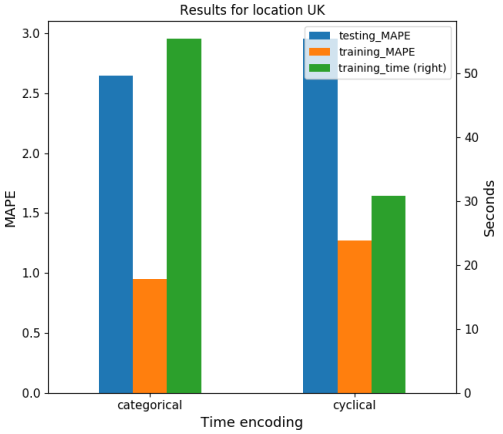


FIGURE 8.5: Time encoding effect on forecasting performance

Other encoding techniques exist, described in the "Implementation" section of this report but were not tested as deemed inferior in most cases by the machine learning community.

These 2 encoding present a trade-off between speed and accuracy, with the cyclical encoding being noticeably faster but yielding lower performance (∼2.95% vs 2.65%). The p-value for both distributions is lower than 0.0002, therefore results are statistically significant.

The cyclical encoding leads to faster training due to lower time input features (11 compared to 73), but it seems that the test model (LSTM) learns better using the categorical encoding (training MAPE is also ∼ 25% lower).

Best performance settings

Time encoding :

- (Faster training) Use cyclical encoding.
- (Better accuracy) Use categorical encoding.

Note : After additional experiments, it was found that the time encoding also noticeably affects the ideal learning rate. A better learning rate for the UK dataset with categorical encoding is 0.002 instead of 0.005 (see "Learning rate" Section below).

8.1.1.4 Machine learning settings

Optimizer

Gradient descent optimizers, described in Section 4.3.3 are the algorithms in charge of determining how to update the weights.

On the problem of load forecasting and using an LSTM network, 3 optimizers show significantly better performance than the others : RMSprop, Nadam and Adam. These similar results aren't surprising since the Adam and NAdam optimizers are variations of the RMSprop optimizer.

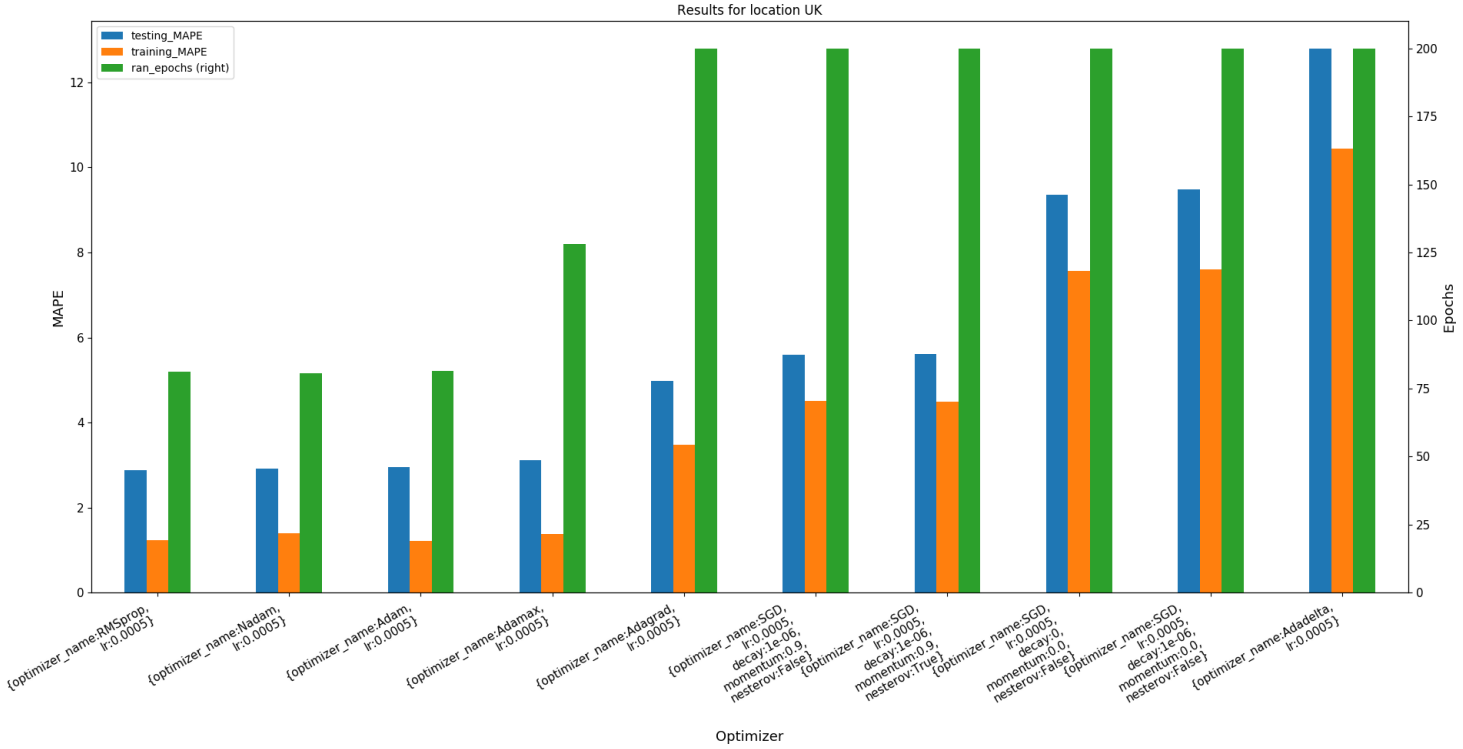


FIGURE 8.6: Neural network optimizer effect on forecasting performance using an RNN model

After 6 runs, the RMSprop optimizer proved to be statistically better than both Adam and Nadam (ttest : $p < 0.03$, ANOVA : p¡0.18), with a mean testing MAPE of ∼2.85% compared to ∼2.93% and ∼2.94% for Adam and Nadam respectively.

Results for CNN models are similar, however the RMSprop optimizer is less efficient than both Adam ($p \approx 0.066$) and Nadam ($p \approx 0.068$) with an average testing MAPE of ∼2.94% versus ∼2.86% respectively, which are indistinguishable ($p \approx 0.45$).

The other optimizers (SGD and other Adam variants) all offer lower accuracy and slower convergence rates. Indeed, all (except Adamax) were not interrupted by the early stopping mechanism : even after 200 epochs, improvements were still being made, albeit extremely slowly and without reaching the performance of other optimizers.

Best performance settings

Optimizer :

- (RNN) Use RMSprop. RMSprop derived optimizers are also good.
- (CNN) Use [N]adam.

Learning rate

The learning rate defines how fast the weights of our networks are updated, its importance is more broadly described in Section 4.3.3.

Typically, only a rather narrow range of learning rates will yield the best performance. This is clearly visible on Figure 8.7 (leftmost figure), where a very low learning rate leads to extremely long training time and reduces performance. On the other end, a learning rate too great will too have slow convergence due to large oscillatory[1] updates.

The more suitable range of values is shown on the rightmost figure of Figure 8.7 where all values are statistically equivalent.

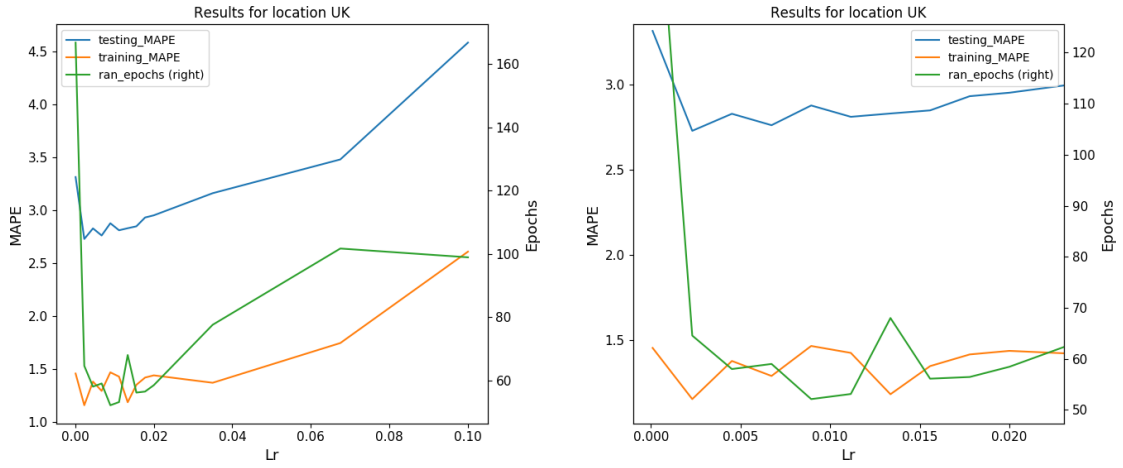


FIGURE 8.7: Learning rate effect on forecasting performance

Best performance settings

Learning rate : Use a learning rate of ~0.005 for the cyclical encoded dataset. Use a lower learning rate when using more features (e.g. categorical encoding).

[1] With regards to the loss function

Early stopping

The early stopping mechanism, briefly described multiple times in this report is used to reduce overfitting and training to the bare minimum while providing optimal testing performance or close to.

More information on early stopping can be found at Prechelt [1998] along with mathematical formulations of the ideal patience parameter, more formally called stopping criterion. In this project, the patience parameter was set to $\sim 1/5$ the number of epochs it takes a model (LSTM) to obtain most of its testing accuracy ($\sim$100 epochs) : 20.

Using a sub-optimal stopping criterion (e.g. stopping too early) can hurt accuracy significantly.

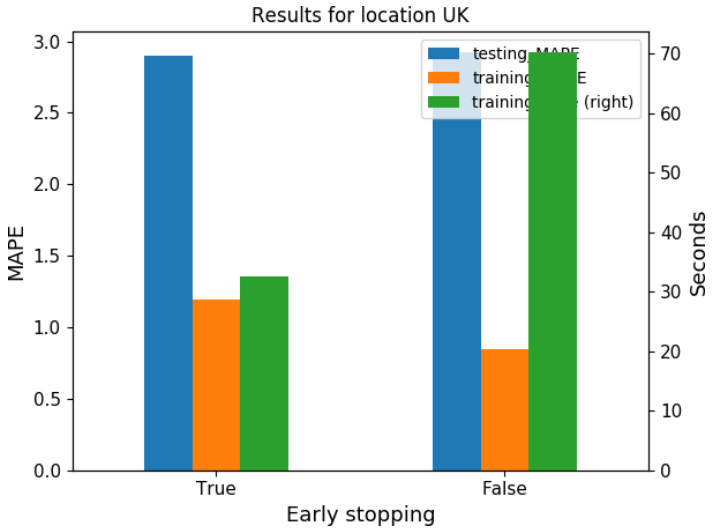


FIGURE 8.8: Early stopping effect on forecasting performance

On our test setup, early stopping clearly reduces training time while providing similar testing MAPE (better mean but not statistically significant after 6 runs : $p \approx 0.57$). We can also see some overfitting when early stopping isn't active as the training and testing accuracy gap is wider than when using early stopping.

Best performance settings

Learning rate : Use early stopping with a large enough patience setting.

Reduced learning rate on plateau

When the loss stops decreasing, it can be due to 2 reasons, either the learning rate is too high and the optimizer keeps bouncing of the sides of a minima (see Figure 4.8), preventing convergence; or the model is already stuck in a local extremum. Thus, the automatic reduction of the learning rate when hitting a plateau aims at mitigating the first scenario.

FIGURE 8.9: Time encoding effect on forecasting performance

The results are statistically significant (6 runs) and shows that the mechanism speeds up convergence ($p \approx 0.071$) but decreases testing performance ($p \approx 0.005$) from ~2.8% to ~2.9% MAPE.

Note that the reduced learning rate will naturally decrease the speed at which the loss decreases (if it decreases), hence may trigger the early stopping mechanism too early.

As specified previously, some optimizers - such as the module's default Adam - include a similar adaptive learning rate.

Best performance settings

Reduced learning rate on plateau :

- (Faster training) Reduce learning rate on plateau
- (Better accuracy) Do not reduce learning rate on plateau

Training epoch count

The number of training epochs is difficult to estimate, yet has a very direct impact on accuracy and training time. To conduct this experiment, early stopping was disabled to ensure the maximum number of epochs is the actual number of epoch run.

Training time grows linearly with epoch count, therefore a large epoch count will result in long training time. Another problem that arises is overfitting, where the training loss gets lower and lower yet the testing loss plateau or increases.

A low epoch count results in very fast training and underfit model which didn't have enough time to learn from the training data, leading to lower accuracy. However, this trade off can sometime be exploited, when computational resources are scarce for example.

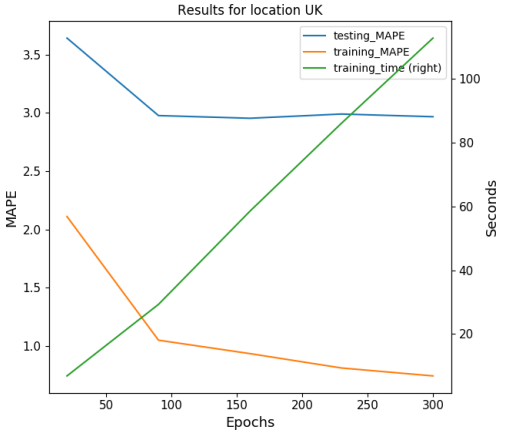


FIGURE 8.10: Training epochs count effect on forecasting performance

It is clear that training for less than ∼90 epochs prevents our model from giving maximum accuracy, while training further only increases training time and decreases training MAPE (which we don't really care about). However, this over-training doesn't yet results in strong overfitting, where the testing accuracy would significantly degrade.

Note that early stopping is meant to fight this very problem by finding the moment when additional training doesn't yield any improvements on the validation set. This, in turn allows for models to be train faster, with optimal performance in most cases.

Best performance settings

Epoch count : Train for ∼100 epochs maximum or use early stopping.

Batch size

The optimizer batch size determines the number of training sample that are aggregated to compute the (estimated) loss. This principle is described in more details in Section 4.3.3. A large batch size results in less back-propagation calculations, therefore faster training but it also hurts performance when the average loss isn't representative of the individual loss of each batch samples. Larger batch size also require linearly more memory to store intermediate results, which can results in slower training. A smaller batch size updates the model's weights more accurately but makes training slower.

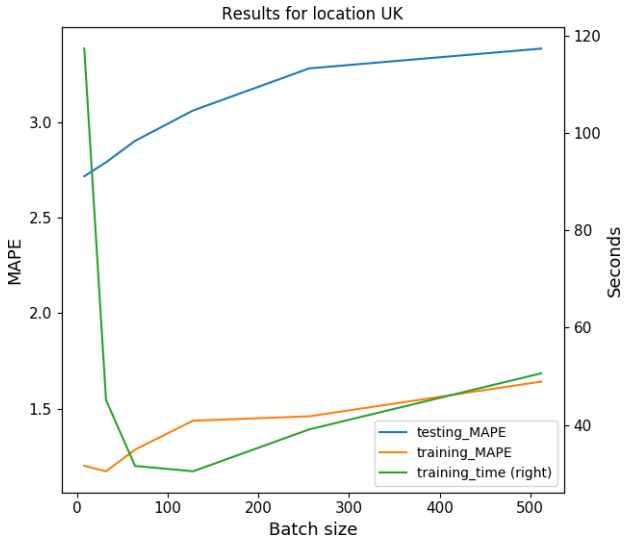


FIGURE 8.11: Batch size effect on forecasting performance

A batch size of 64 offers fast training with great accuracy while a batch size of 32 increases both training time and accuracy. A batch size of 8 further decreases the MAPE but training slows down significantly.

- [8] : ∼2.7% MAPE and ∼4 training time units.
- [32] : ∼2.8% MAPE and ∼1.5 training time units.
- [64] :∼2.9% MAPE and ∼1 training time unit.

A batch size of 64 yields ∼2.9% MAPE and training time of 1 (unit), while 8 yields

Best performance settings

Batch size :

- (Faster training) Use batch size of 64.
- (Better accuracy) Use batch size of 32.
- (Best accuracy) Use batch size of 8.

8.1.1.5 Environment settings

Training device (i.e. GPU vs CPU)

GPUs have thousands of low-complexity cores, offering high throughput with parallel problems, such as training Deep Learning models as shown on Figure 8.12. For more information, an in-depth discussion can found at Quora - Why are GPUs well-suited to deep learning?.

Inference is also faster on GPUs but being a much less intensive task, a CPU is sufficient. Furthermore, inference in a production environment is typically done on low power machines, without dedicated GPUs.

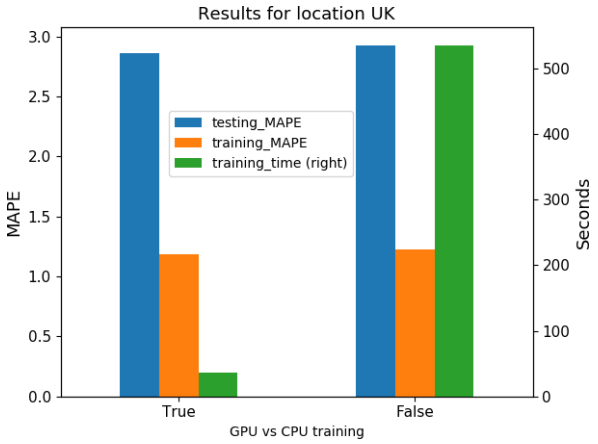


FIGURE 8.12: Training device impact on forecasting performance

Training performance on a modern high-power laptop CPU (Intel i7-7700HQ) is approximately 10 to 15 times slower than training using a mid-range laptop GPU (Nvidia GTX 1060 Max-Q). However, accuracy metrics are similar.

Best performance settings

Training device : Use a GPU, really.

Keras backend

Keras is a high-level neural network API. As such, it doesn't implement low level operations but instead relies on other framework such as Tensorflow, CNTK[1] and Theano.

Performance testing using Tensorflow and CNTK backend are visible in Figure 8.13.

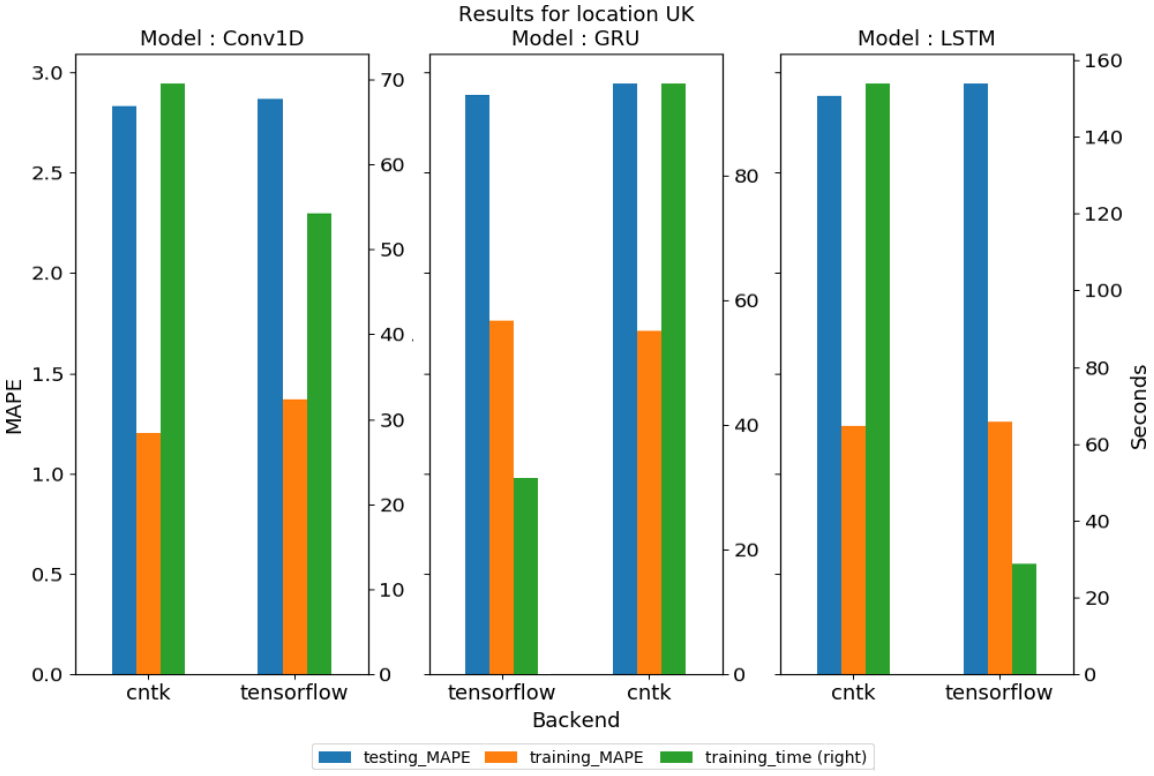


FIGURE 8.13: Backend impact on performance for CNN, GRU and LSTM models

CNTK doesn't support the CuDNN[2] accelerated versions of the LSTM and GRU models, this could be the main reason behing the significantly slower training. Accuracy is on par though.

When it comes to the test CNN model, differences are not statistically significant.

Dense models were not tested because of their lower accuracy and long training time.

Overall, the results are quite clear, despite CNTK being capable to build accurate models, training can be significantly slower, hence Tensorflow is a safer option.

Best performance settings

Keras backend : Use Tensorflow backend.

[1] Microsoft's CogNitive ToolKit.
[2] Nvidia CUDA Deep Neural Network library - providing highly optimized neural network operations.

Floating point precision

Neural network parameters can be stored as 64 bits ('FP64') or 32 bits ('FP32') floating point numbers and can affects the model's performance. Greater precision (64 bits) can lead to better performance but at the cost of reduced training speed and doubled memory usage.

Most modern GPUs natively perform single precision operations (FP32) and offer emulated double precision mode with a theoretical 1/32 and 1/4 to 1/16 throughput for consumer grade Nvidia and AMD GPUs, respectively. However, real life applications typically do not experience such large slowdowns. F64 and F16 are natively available on "scientific-grade" GPUs and can be useful for high-precision simulations or faster low precision computation.

Results using the default LSTM model are visible in Figure 8.14.

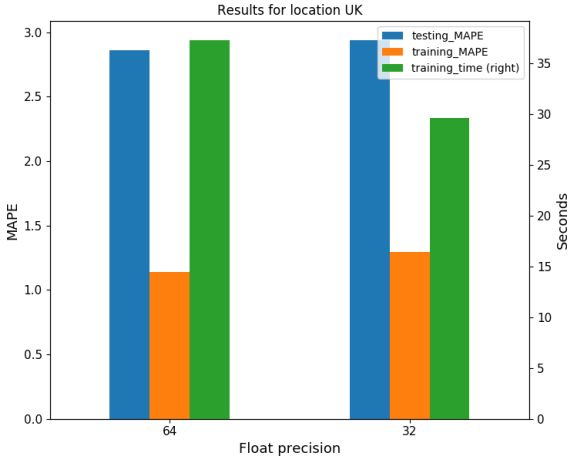


FIGURE 8.14: Floating point precision impact on forecasting performance

In our experiment, the described trade-off is visible, with 64 bit giving lower testing MAPE ($p \approx 0.037$) and significantly lower training MAPE but suffers from slightly longer training times ($\sim 25\%$).

Best performance settings

Floating point precision :

- (Faster training) Use 32 bit precision.
- (Better accuracy) Use 64 bit precision.

8.1.2 Deep learning models architectures

The determination of the optimal architecture for a neural network is currently mainly done via pseudo-exhaustive search[1], as such, a variety of models are compared in this section.

Note that the models were trained with a non-optimal (with regards to accuracy) batch size of 256 to speed up training. Yet, the following graphs were generated from more than 48 hours of computation on the most powerful GPU[2] on the Heriot Watt Robotarium cluster. Keep in mind that those results are the average of 3 runs only, as such, close results may not be statistically different.

Previous experiments were run on a different hardware, as such the training time cannot be directly compared - yet the determined best settings are believe to be true under most conditions (in the scope of this project).

8.1.2.1 Recurrent Neural Networks

Recurrent models are particularly suited for time series data as they possess some internal memory allowing for time dependent predictions (i.e. 2 consecutive identical input samples won't necessarily generate the same output).

This is akin to feeding output data back as input ("Vanilla RNN") but using a more complex setup in order to combat overfitting and improve performance for example. More details can be found in Section 4.3.3.2.

Both LSTM and GRU (Gated Recurrent Unit) are tested. GRU are modern variant of the LSTM cells and feature one less internal layer, hence are theoretically more computationally efficient. Accuracy should be on par with LSTM models.

Both models are built using the following template :

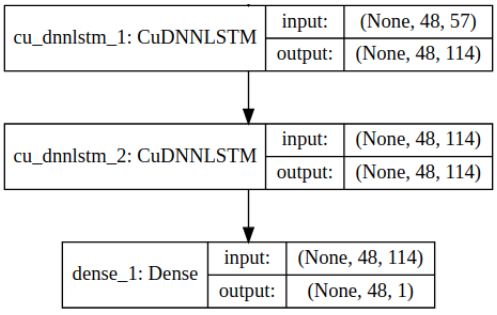


FIGURE 8.15: RNN test structure

[1] Testing of many combinations, the search space being infinite.
[2] All models do not leverage all available 'power' due to limited algorithmic parallelism.

The input and output shapes of each layer are interpreted as such : An undefined "None" number of data "blocks" of 48 rows ('records' - one per 30 minute) by 57 columns (input features). The last layer's output is (None, 48, 1) as it outputs the next 48 30 minutes forecast load (1).

The attentive reader will see that the number of input neurons is 57 instead of 56 in previous benchmarks, this is due to a missing input feature in older datasets : H-24 load - this difference doesn't effect any of the conclusions drawn in this report.

FIGURE 8.16: RNN model structure performance

GRU and LSTM being similar architectures, the accuracy results aren't very different. It is clear that using more units per layer improves performance, without impacting training speed as more of the parallel hardware is used.

When it comes to the number of layers, for LSTM, 4 layers seems to offer the best performance while GRUs require a few extra layer to reach a comparable accuracy.

Speed-wise, the GRU being a simpler architecture than the LSTM, it tends to train faster with training times ranging from 30 to 60 seconds compared to the longer LSTM training times from 30 to 90 seconds, regardless of the layer count.

Interestingly, both model types train faster (∼10%) when the number of hidden units is 192 compared to models with slightly more of less units.

Best performance settings

RNN structure :

- (Layer count) Use between 3 and 5.
- (Unit per hidden layer) Use ∼200 neurons or more.
- (Cell type) Undetermined, very similar performance in this experiment.

8.1.2.2 Convolutional Neural Networks

Convolutional Neural Networks (CNN) are typically used for images (2D data), but 1D convolutions are also possible - these models are explored in this section.

The basic CNN structure used in those test is identical to that of the RNNs (Figure 8.15), with all layers replaced by Convolution1D layers along with a fixed kernel size of 3 for all layers (see 4.3.3.3 for a reminder of what CNN kernels are).

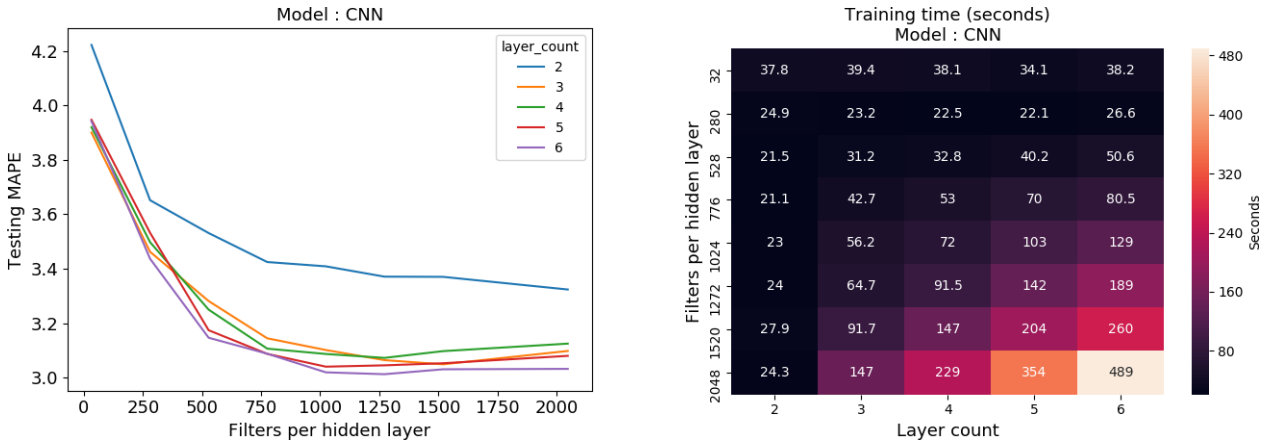


FIGURE 8.17: CNN model structure performance

Convolutional neural networks with a single hidden layer (i.e. not Deep Learning) clearly show lower performance but adding more than 1 additional layer doesn't yield significant performance improvements, though models with 5 and 6 layers perform slightly better. However, training time increases quickly as the number of neurons and layers increases.

The last layer of the model can be either a 1D Convolutional layer or a typical dense layer. There is not statistical better choice when using the data from 15 runs, as visible on Figure 8.18. Results are from test models with 4 layers and 5 different filter counts per layer.

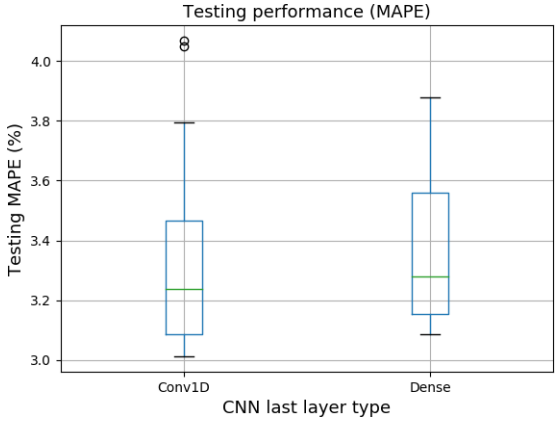


FIGURE 8.18: CNN last layer effect on performance

Overall, this experiment shows that CNN are capable to perform very well on this task but require rather long training time which quickly increases when adding more layers and filters per layer.

The best CNN models scored a testing MAPE of $\sim 3.01\%$ after $\sim$190 seconds of training - the 6 layers, 1272 filters network which is significantly slower than both LSTM and GRU, but with lower MAPE.

A noteworthy model is the 3 layers, 1277 filters network with a MAPE of $\sim 3.06\%$ and only $\sim$65 seconds of training, which is similar to best performing RNN models.

Best performance settings

CNN structure :

- (Layer count) Use 3 or more layers.
- (Filter per hidden layer) Use around 1024 filters.
- (Last layer) Undetermined, very similar performance in this experiment.

8.1.2.3 Dense Neural Networks

Dense models are the most basic types of neural networks, as such they do not benefit from any advanced features such as recurrence or overfitting mitigation.

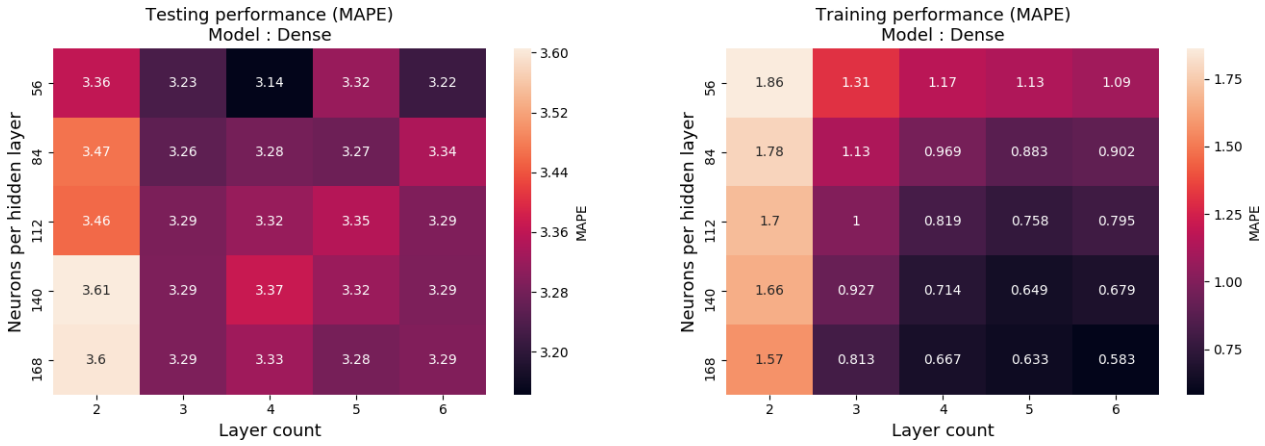


FIGURE 8.19: Dense model structure performance

The Dense architecture testing accuracy results are surprising at first glance, with the added neurons per layer and added layers mostly decreasing performance. However, when looking at the training performance : our models are strongly overfiting.

Note that these benchmarks were done with identical load propagation settings as for RNN and CNN, yet Dense models are likely to benefit most from additional near historical data (e.g. H-4) - however this would likely have increased overfitting and training time - both of which would make Dense models even less compelling when compared to other models.

Due to the above assumption[1]), Dense models are run under the assumption that their input data contains forecast values, hence the testing set needs to be constructed in a "recurrent" manner. As such the early stopping was disabled because it only accepts a single fixed dataset. To limit overfitting and reduce training times, the number of maximum epochs was set to 100 as opposed to 200.

The training of these models is extremely slow in comparison to other discussed models, at $\sim$ 160 seconds for 1 layer models up to $\sim$ 250 seconds for 6 layer models. The number of neurons doesn't affect training speed at all, due to the large parallel processing power of the GPU on which they were training.

Best performance settings

Dense structure :

- (Layer count) Use only few layers to avoid overfitting.
- (Unit per hidden layer) Use fewer neurons to avoid overfitting.
- (General) Do not use Dense models.

[1] Which was not validated nor invalidated, due to time constraints and because Dense models offer lower performance than other models.

8.1.3 Best models comparison

Now that we have a good idea of the optimal settings to use for the whole processing pipeline and training settings along with good network architectures, we can make a serious attempt at tackling the problem of load forecasting. Settings leading to best accuracy were used, except for the batch size, set to 64 in order to make training reasonably fast.

The following models will be compared :

- (GRU) 4 layers with 2X units per hidden layer
- (GRU) 4 layers with 4X units per hidden layer
- (LSTM) 5 layers with 2X units per hidden layer
- (LSTM) 4 layers with 4X units per hidden layer
- (CNN) 5 layers with 1024 filters per hidden layers
- (CNN) 3 layers with 1024 filters per hidden layers

'X' refers to the number of input neurons.

8.1.3.1 NYC dataset

Figure 8.20 show the performance results for the NYC dataset.

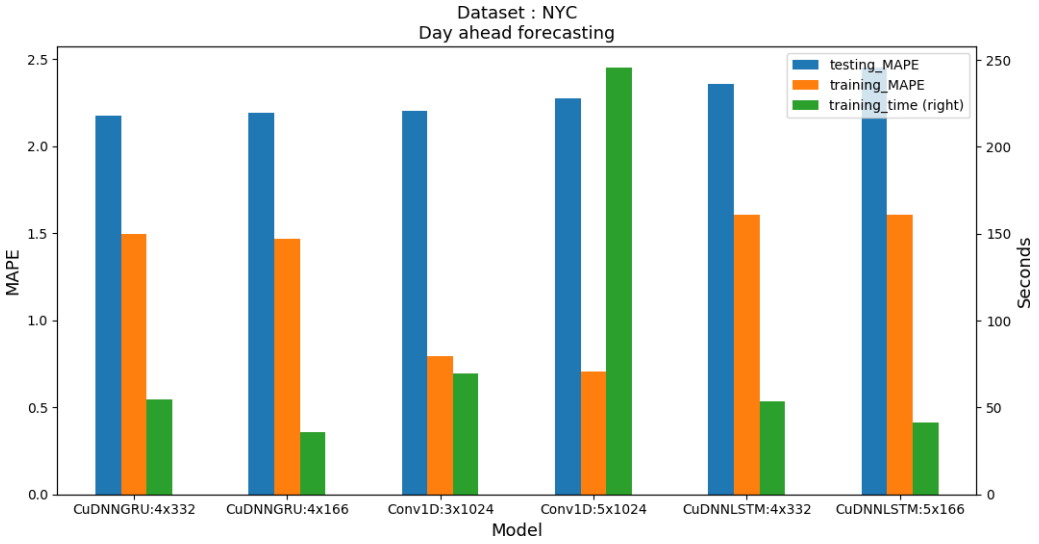


FIGURE 8.20: Best performing models on the NYC dataset

Both GRU models as well as the smaller CNN yield good performance with statistically non significant accuracy differences, yet the GRU based models trained faster than the CNN model.

However, the bigger CNN models suffers from very long training time, while underperforming on the test set. This is due to its large number of trainable parameters (∼10 million) compared to ∼3.4 million for the smaller CNN, down to ∼450,000 for the smaller RNN.

CNN are capable of impressive training set accuracy as such, adding some regularization such as dropout should be beneficial - unfortunately this possibility won't be explored in the scope of this project.

In this setup, we can see the GRU pulling ahead of his older cousin, the LSTM.

Overall, the 4 layers (166 hidden units) GRU model performs best at ∼2.2% testing MAPE and with a reasonable training time of ∼ 50 seconds.

These results clearly beat the NYISO forecasts which averages at ∼3.25% MAPE !

A distribution of NYISO error and the GRU:4x322 model can be seen in Figure 8.21.

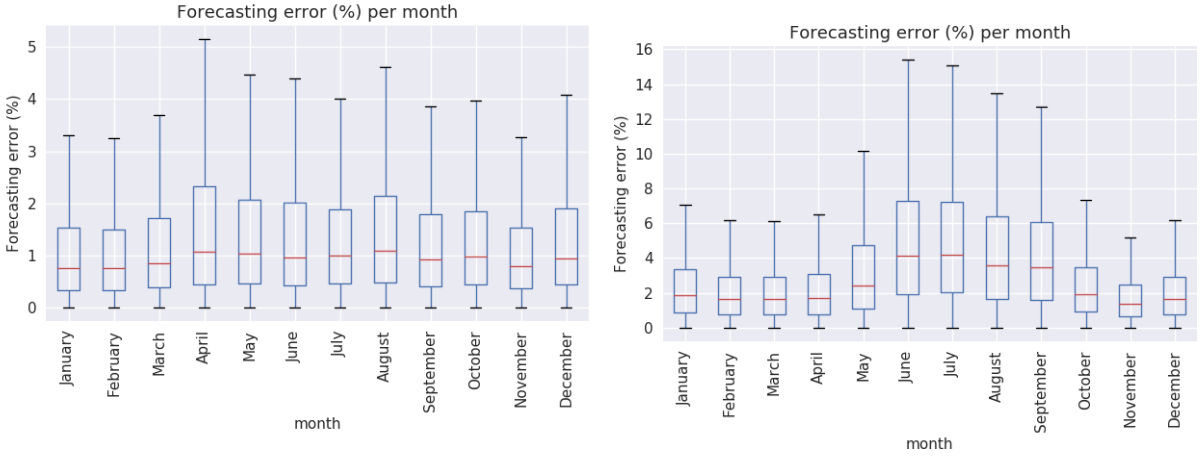


FIGURE 8.21: Error distribution of NYISO and GRU:4x322 on the NYC dataset

The NYISO forecasting model clearly struggle with summer data and while our model does too, it does so to a much lower degree.

8.1.3.2 UK dataset

Figure 8.22 show the performance results for the UK dataset.

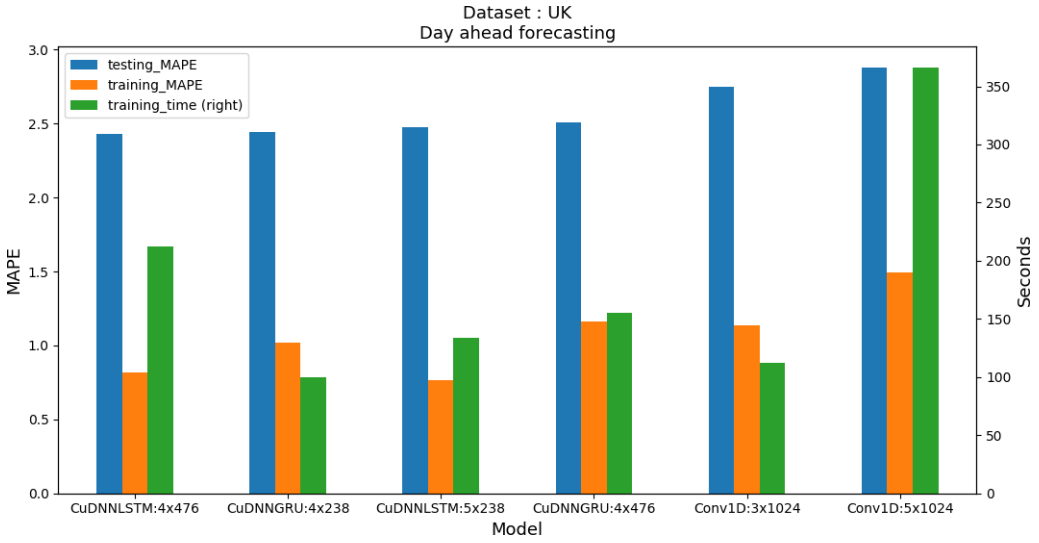


FIGURE 8.22: Best performing models on the UK dataset

In this test, the large CNN is last, according to all metrics and by a significant margin. The best model is the LSTM 4x476 (4 layers, 476 units per hidden layer) but the GRU 4x238 is nearly as accurate yet twice as fast. The accuracy difference is not statistically significant ($p \approx 0.43$).

Once again, the GRU based model outperformed[1] the others models on this dataset, with a MAPE of ~2.4% and a training time of ~100 seconds, at least 30% faster than other tested models on this dataset.

We can note the large difference in training time between the 2 CNNs, yet the smaller one performed better - but is unable to beat the RNN models.

[1] When taking into consideration both trainin speed and testing accuracy

8.1.3.3 MHK dataset

Figure 8.23 show the performance results for the UK dataset.

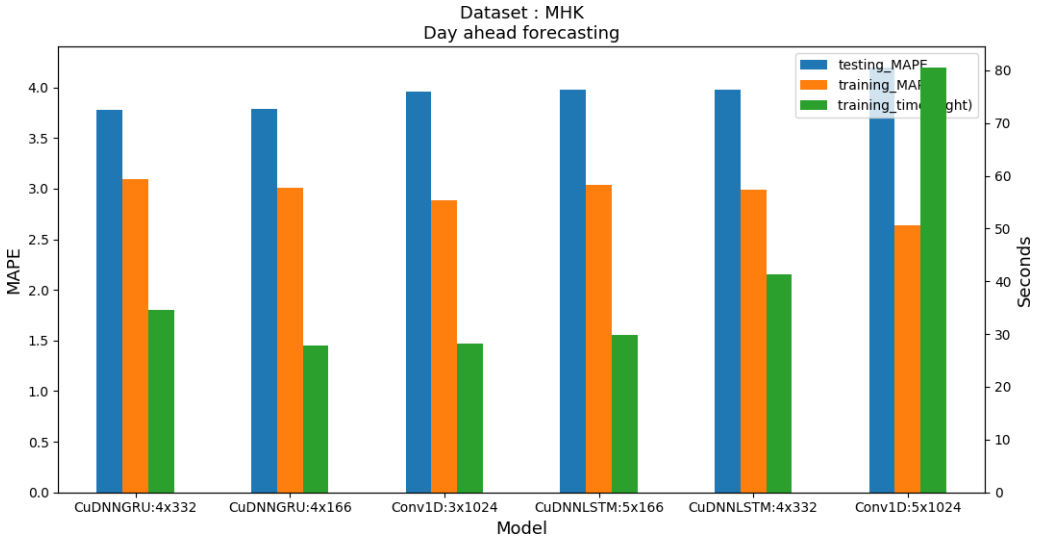


FIGURE 8.23: Best performing models on the MHK dataset

Yet again, the smaller (166 units) 4 layer GRU is the best model. Notice that the training time is only 30 seconds, 3 times faster than on the UK dataset. This is because the dataset contains a lot of noise that the model cannot capture on the test set, hence the validation loss doesn't decrease (it actually jitters a lot and increases) therefore, the early stopping mechanism kicks in and stops training. Note that the model is still capable to get low training error (lower than what is shown on 8.23) but cannot generalize well.

The CNN is again unable to reach the level of performance of the GRU models, but the smaller one outperforms the LSTM models.

The monthly error distribution for the GRU:4x166 and NYISO are visible on Figure 8.24
These 2 plots would be clearer if on the same Figure, but notice how the error scale is very different : our model has much lower MAPE error than the NYISO forecasting system. The error profile is similar otherwise, with peak errors during the summer.

Despite the lower performance compared to other datasets; all 4 models are much, much better than the NYISO proposition of ∼16% average MAPE.

On Figure 8.25, a load forecasting plot comparing the true load value over a random period in the testing data against our model GRU:4x166 and NYISO forecasts. While this Figure doesn't carry any statistical significance, it gives some perspective to MAPE metrics.

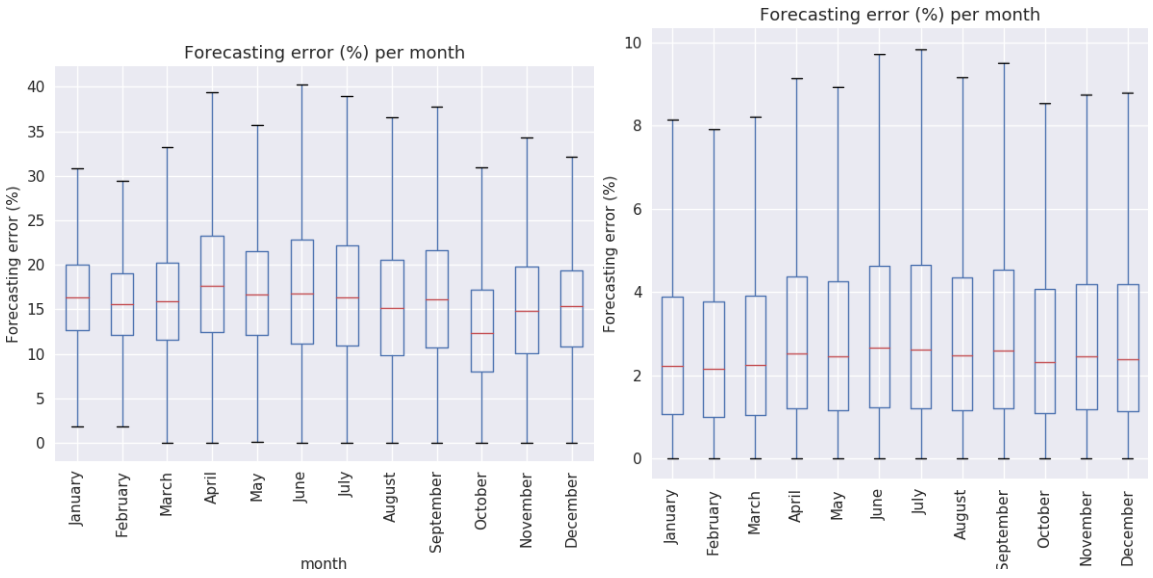


FIGURE 8.24: Error distribution of NYISO and GRU:4x166 on the NYC dataset

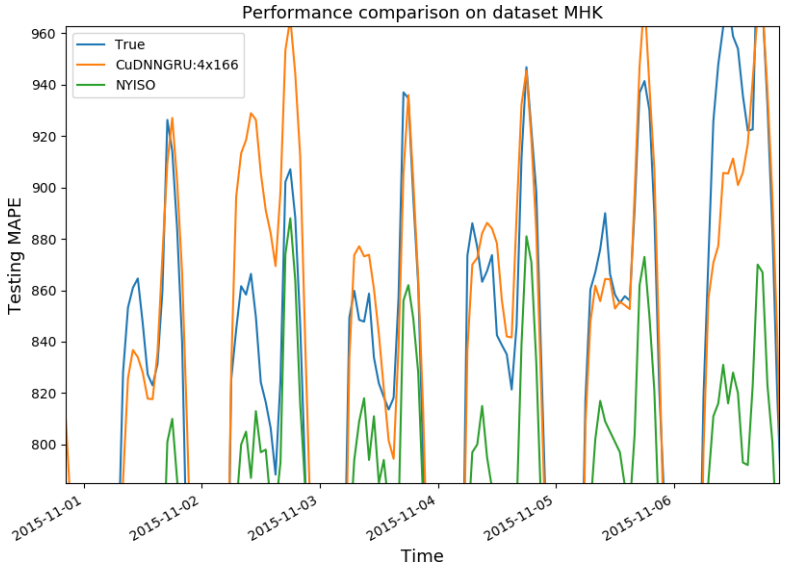


FIGURE 8.25: Load forecasting GRU:4x166 vs NYISO vs ground truth

8.1.4 General conclusion

The best model on all datasets was the smaller 4 layer GRU architecture with testing MAPE of ∼2.2%, ∼2.4% and ∼3.8% on the NYC, UK and MHK datasets respectively - beating the NYISO forecasts on both NYC and MHK datasets by a significant margin (at ∼3.25% and ∼16% MAPE).

Note that despite all these experiments, better models and settings can likely be found by iterative optimization : identifying a best parameter x for variable X, running all other tests with new x, check that x is still the best value for variable X etc... Also, experiments were run on the dataset with cyclical time encoding, yet it turned out that categorical encoding leads to better accuracy. As such, the previous experiments (variables exploration) may be less accurate when using the different encoding.

Also, the models architectures can further be tweaked by varying the number of units per layers independently from one another or using more advanced models such as stateful and bidirectional LSTM, Recurrent CNN, adding dropout layers etc..

Lastly, the model training scheme can be improved, for example, by using a cyclical learning rate and weight averaging as described in Izmailov et al. [2018].

Results evaluation

To evaluate the models, the Mean Average Percentage Error (MAPE) was used primarily as it enables the forecasting performance comparison between datasets, in contrast with Root Mean Square Error (RMSE) which is scale-dependant.

Unfortunately, no relevant paper dealing with either NYISO or National Grid UK electricity data was found. However, NYISO provides forecasts to which the models could be compared : the developed models outperformed the NYISO forecasts on both datasets.

For the UK dataset however, no point of reference other than the typical range of values found in the literature on other datasets was used.

Note that on all datasets, the testing performance noticeably increased when using settings identified in Section 8.1.1 compared to the module defaults used in the variable exploration phase.

A lot of metrics were captured, more than what is shown in this report. Example of metrics include : mean GPU usage, number of training parameters, model's memory usage as well as inference time for both train and test sets.

All results from this section are available in Table 8.1 as well as in a NoSQL database (along with results from previous sections) which will be shared with the relevant stakeholders of this MSc project. Plotting methods and additional graphs will also be available within the project's submission archive.

Dataset	Model architecture	Testing MAPE	Testing RMSE	Training MAPE	Training RMSE	Training time
MHK	Conv1D:3x1024	3.96 %	41.36	2.89 %	30.96	28 secs
	Conv1D:5x1024	4.19 %	43.87	2.64 %	28.52	80 secs
	CuDNNGRU:4x166	**3.78 %**	**39.71**	**3.01 %**	**32.22**	**28 secs**
	CuDNNGRU:4x332	3.78 %	39.72	3.09 %	32.92	35 secs
	CuDNNLSTM:4x332	3.98 %	41.76	2.98 %	31.98	41 secs
	CuDNNLSTM:5x166	3.97 %	41.79	3.04 %	32.64	30 secs
	NYISO	16.09 %	137.78	N/A	N/A	N/A
NYC	Conv1D:3x1024	2.20 %	161.72	0.80 %	58.65	69 secs
	Conv1D:5x1024	2.28 %	165.60	0.71 %	52.37	245 secs
	CuDNNGRU:4x166	**2.19 %**	**164.89**	**1.47 %**	**111.42**	**36 secs**
	CuDNNGRU:4x332	2.17 %	163.71	1.50 %	112.69	54 secs
	CuDNNLSTM:4x332	2.36 %	176.02	1.60 %	120.44	53 secs
	CuDNNLSTM:5x166	2.45 %	182.61	1.61 %	120.03	41 secs
	NYISO	3.26 %	318.29	N/A	N/A	N/A
UK	Conv1D:3x1024	2.75 %	1039.62	1.14 %	476.07	112 secs
	Conv1D:5x1024	2.88 %	1091.45	1.50 %	630.38	366 secs
	CuDNNGRU:4x238	**2.44 %**	**926.31**	**1.02 %**	**419.18**	**100 secs**
	CuDNNGRU:4x476	2.51 %	948.05	1.16 %	480.40	155 secs
	CuDNNLSTM:4x476	2.43 %	921.65	0.82 %	338.37	213 secs
	CuDNNLSTM:5x238	2.47 %	937.40	0.76 %	316.40	134 secs

TABLE 8.1: Day ahead load forecasting: Summary of results

Chapter 9

Conclusion

In this MSc project, a load forecasting module was implemented using the Python programming language using the Keras library.

As many variables influence the load forecasting process, the software exposes settings that can easily be modified through the software API[1] and their effects have been explored along with the comparison of multiple Deep Learning neural network architectures.

The system was tested on the task of day ahead load forecasting on 3 datasets : NationalGrid's UK, and 2 datasets from the NYISO on which the developed models were able to outperform the NYISO forecasts. Performance on the National Grid dataset is harder to assess due to the lack of relevant papers and forecasting data, but models performed within the typical range of error found in the literature.

The Recurrent Neural Network models and more specifically, the Gated Recurrent Unit (GRU) based model consistently delivered high quality forecasts (∼2.2 - 3.8% MAPE) with a limited training budget (30-100 seconds)

Overall, the project succeeded in covering all mandatory requirements as well as some optional ones including higher precision forecasts and to some extent, re-usability which allows the system to be used in settings it was not originally intended for, with little code modification.

In conclusion, this project showed that the electricity load / demand forecasting problem can be addressed using Deep Learning methods, including Recurrent and Convolutional Neural Network. This project also highlights the importance and complexity that lies within the data pre-processing process.

[1] Application Programming Interface

9.1 Future work

9.1.1 Machine learning engineering

The field of Machine learning being as large and complex as it is, there are many ways to improve the forecasting accuracy.

Examples include using more advanced models, such as those described in the literature review, using ensemble and weight averaging methods or use differenced time series as in Kuremoto et al. [2014]'s work.

Also, an SVR model is implemented but isn't yet interfaced with the top level API of the module, which would be a nice addition to the forecasting module.

Different forecasting horizon could be assessed but reliable external forecasts dataset need to be found as to make evaluation more relevant. Note that the module already supports week-ahead forecasting.

9.1.2 Software engineering

From a software standpoint, the module being a rather complex piece of code developed in a short period of time, some form of refactoring could be beneficial, mainly to improve error handling and overall stability.

Performance wise, multiple section have already been highly optimized, yet a lot of improvements can be made such as using tools to leverage parallel hardware during the data pre-processing step such as Dask. Also, some code sections can also be further optimized and possibly compiled using a Just-In-Time (JIT) compiler such as Numba.

Improved operation feedback through the through the use of the Python logging module and exceptions would ease debugging and adoption by the community.

9.1.3 Further results analysis

A lot of metrics were collected, and a complete analysis requires a lot of time but can bring insight to further improve the module's performance.

9.1.4 End user usage

The addition of a Graphical User Interface as well as a REST API would help this project have a greater impact in the data science space.

Appendix A

Weather data acquisition

The weather datasets (for the UK and New York state) can be requested from NOAA's[1] National Climatic Data Center (NCDC), using the following procedure :

1. (a) Go to NCDC Integrated Surface Database.
   (b) Select "ADVANCED".
   (c) Select country (and state if relevant).
2. (a) Find the main HTML table within the HTML source code of the page (e.g. using the web browser's developer tools).
   (b) Convert stations list to csv using an HTML-to-csv tool such as Online HTML-to-csv. Remove the first row of the HTML table before converting.
3. (a) Replace multiple spaces with single spaces using a text editor's "Find & Replace" for example.
   (b) Wrap stations names (can contain spaces) with quotation marks using :

   ```
   '^'' -> '" and '..' -> '.."'
   ```

   (c) Replace 'to ' with "
   (d) Add csv header 'station_name station_code start end'.
4. (a) Load data in Python using the pandas library and the following code

   ```
   import pandas as pd
   pd.read_csv(FILENAME, sep= ' ')
   ```

   (b) Convert date columns into Python date type using the following code :

[1] NOAA : National Oceanic and Atmospheric Administration

```
data['start'] = pd.to_datetime(data['start'])
data['end'] = pd.to_datetime(data['end'])
```

(c) Filter by date using the following code :

```
start = pd.Timestamp(StartYear, StartMonth, StartDay)
end = pd.Timestamp(EndYear, EndMonth, EndDay)
IDX = data[data['start'] <= start & data['end'] >= end].index
```

(d) Get stations indices using "data.index"

5. (a) Open web browser console and copy the stations indices IDX from Python using :

```
idx = <IDX>
```

(b) (Using Chrome) Select the HTML table containing the weather stations.

(c) Select the stations from the table using the following Javascript code in the browser console :

```
for (var i = 0; i < idx.length; i++) {}
    $0.rows[2].cells[0].getElementsByTagName('option') \
    [idx[i]].selected = 1 }
```

6. Select variables "Air Temperature Observation", "Computed Relative Humidity" and "Wind Observation".

7. Select dates (same as in step '4.(c)').

8. Check your inbox a few hours later and download the stations file and data file.

9. Modify the downloaded files. **Yes, this should be scripted. Maybe it will.**

(a) Replace the two first lines of the data file with the following "USAF,NCDC,Date,HrMn,I,Type,QCP,W_Dir,W_Q,W_I,W_Spd,W_Q2, Air_Temp,Air_Q,Dew_temp,Dew_Q,RHx". Remove the field 'NCDC' for New York data.

(b) Remove the second line from the stations file. (dashed line "———[...]")

(c) Replace field separators (multiple spaces) by single spaces.

(d) (For New York only - stations file) Replace the header field "USAF-WBAN_ID" by "WBAN USAF". Replace "STATION NAME" with "STATION_NAME". If necessary, enclose stations names, country and state into double quotes.

Bibliography

Abraham, A. (2005). *Artificial Neural Networks*. John Wiley and Sons, Ltd.

Ackley, D. H., Hinton, G. E., and Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169.

Amarasinghe, K., Marino, D. L., and Manic, M. (2017). Deep neural networks for energy load forecasting. In *Industrial Electronics (ISIE), 2017 IEEE 26th International Symposium on*, pages 1483–1488. IEEE.

Bashir, Z. and El-Hawary, M. (2009). Applying wavelets to short-term load forecasting using pso-based neural networks. *IEEE transactions on power systems*, 24(1):20–27.

Box, G. E. and Cox, D. R. (1964). An analysis of transformations. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 211–252.

Debnath, K. B. and Mourshed, M. (2018). Forecasting methods in energy planning models. *Renewable and Sustainable Energy Reviews*, 88:297–325.

Dedinec, A., Filiposka, S., Dedinec, A., and Kocarev, L. (2016). Deep belief network based electricity load forecasting: An analysis of macedonian case. *Energy*, 115:1688–1700.

Dedinec, D. (2016). Correlation of variables with electricity consumption data. *Eventiotic*.

Dong, B., Li, Z., Rahman, S. M., and Vega, R. (2016). A hybrid model approach for forecasting future residential electricity consumption. *Energy and Buildings*, 117:341 – 351.

Drezga, I. and Rahman, S. (1999). Phase-space based short-term load forecasting for deregulated electric power industry. In *Neural Networks, 1999. IJCNN'99. International Joint Conference on*, volume 5, pages 3405–3409. IEEE.

Drucker, H., Burges, C. J. C., Kaufman, L., Smola, A. J., and Vapnik, V. (1997). Support vector regression machines. In Mozer, M. C., Jordan, M. I., and Petsche, T., editors, *Advances in Neural Information Processing Systems 9*, pages 155–161. MIT Press.

Dudek, G. (2015). Short-term load forecasting using random forests. In *Intelligent Systems' 2014*, pages 821–828. Springer.

Eberhart, R. and Kennedy, J. (1995). A new optimizer using particle swarm theory. In *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on*, pages 39–43. IEEE.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231.

Federal-Energy-Regulatory-Commission (2018). Nysio zones map.

Financial-Times (2017). Deepmind and national grid in ai talks to balance energy supply.

Fukunaga, K. and Hostetler, L. (1975). The estimation of the gradient of a density function, with applications in pattern recognition. *IEEE Transactions on information theory*, 21(1):32–40.

Hao, Z. (2017). Activation functions in neural networks.

Hassan, S., Khosravi, A., and Jaafar, J. (2013). Neural network ensemble: Evaluation of aggregation algorithms in electricity demand forecasting. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–6. IEEE.

Haykin, S. S. (2009). *Neural networks and learning machines*. Pearson Education, third edition.

He, W. (2017). Load forecasting via deep neural networks. *Procedia Computer Science*, 122:308–314.

Hernández, L., Baladrón, C., Aguiar, J. M., Calavia, L., Carro, B., Sánchez-esguevillas, A., Cook, D. J., Chinarro, D., and Gómez, J. (2012). A study of the relationship between weather variables and electric power demand inside a smart grid/smart world framework. sensors 2012.

Hernández, L., Baladrón, C., Aguiar, J. M., Carro, B., Sánchez-Esguevillas, A., and Lloret, J. (2014). Artificial neural networks for short-term load forecasting in microgrids environment. *Energy*, 75:252 – 264.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hong, T. (2015). Crystal ball lessons in predictive analytics. *EnergyBiz Mag*, pages 35–37.

Hosein, S. and Hosein, P. (2017). Load forecasting using deep neural networks. In *Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT), 2017 IEEE*, pages 1–5. IEEE.

Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154.

Izmailov, P., Podoprikhin, D., Garipov, T., Vetrov, D., and Wilson, A. G. (2018). Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*.

Joohyoung (2015). Deep learning theory and practice (revised).

Jovanović, S., Savić, S., Bojić, M., Djordjević, Z., and Nikolić, D. (2015). The impact of the mean daily air temperature change on electricity consumption. *Energy*, 88:604 – 609.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

Kuo, P.-H. and Huang, C.-J. (2018). A high precision artificial neural networks model for short-term energy load forecasting. *Energies*, 11(1):213.

Kuremoto, T., Kimura, S., Kobayashi, K., and Obayashi, M. (2014). Time series forecasting using a deep belief network with restricted boltzmann machines. *Neurocomputing*, 137:47–56.

Lahouar, A. and Slama, J. B. H. (2015). Day-ahead load forecast using random forest and expert input selection. *Energy Conversion and Management*, 103:1040–1051.

Marino, D. L., Amarasinghe, K., and Manic, M. (2016). Building energy load forecasting using deep neural networks. In *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, pages 7046–7051. IEEE.

Mathur, S. (2012). Debugging the process of applying a machine learning algorithm to a dataset.

of-Information-and Computer-Sciences, D.-B.-S. (2000). Svm lecture.

Olah, C. (2015). Understanding lstm networks.

Ouyang, T., He, Y., Li, H., Sun, Z., and Baek, S. (2017). A deep learning framework for short-term power load forecasting. *arXiv preprint arXiv:1711.11519*.

Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, pages 55–69. Springer.

Qiu, X., Zhang, L., Ren, Y., Suganthan, P. N., and Amaratunga, G. (2014). Ensemble deep learning for regression and time series forecasting. In *Computational Intelligence in Ensemble Learning (CIEL), 2014 IEEE Symposium on*, pages 1–6. IEEE.

Ramezani, M., Falaghi, H., Haghifam, M.-R., and Shahryari, G. (2005). Short-term electric load forecasting using neural networ@articleruxton2006unequal, title=The unequal variance t-test is an underused alternative to Student's t-test and the Mann–Whitney U test, author=Ruxton, Graeme D, journal=Behavioral Ecology, volume=17, number=4, pages=688–690, year=2006, publisher=Oxford University Press ks. In *Computer As A Tool, 2005. EUROCon 2005. The International Conference on*, volume 2, pages 1525–1528. IEEE.

Redmon, J., Divvala, S. K., Girshick, R. B., and Farhadi, A. (2015). You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640.

Reif, R. (2017). Importance of feature scaling in data modeling.

Ruder, S. (2016a). An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747.

Ruder, S. (2016b). An overview of gradient descent optimization algorithms.

Ruxton, G. D. (2006). The unequal variance t-test is an underused alternative to student's t-test and the mann–whitney u test. *Behavioral Ecology*, 17(4):688–690.

Ryu, S., Noh, J., and Kim, H. (2016). Deep neural network based demand side short term load forecasting. *Energies*, 10(1):3.

Shi, H., Xu, M., Ma, Q., Zhang, C., Li, R., and Li, F. (2017). A whole system assessment of novel deep learning approach on short-term load forecasting. *Energy Procedia*, 142:2791–2796.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676):354.

Smith, L. N. (2017). Cyclical learning rates for training neural networks. In *Applications of Computer Vision (WACV), 2017 IEEE Winter Conference on*, pages 464–472. IEEE.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.

Stanford-University (2017). Convolutional neural networks for visual recognition.

Suykens, J. and Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural Processing Letters*, 9(3):293–300.

Szegedy, C., Ioffe, S., and Vanhoucke, V. (2016). Inception-v4, inception-resnet and the impact of residual connections on learning. *CoRR*, abs/1602.07261.

Tong, C., Li, J., Lang, C., Kong, F., Niu, J., and Rodrigues, J. J. (2017). An efficient deep model for day-ahead electricity load forecasting with stacked denoising auto-encoders. *Journal of Parallel and Distributed Computing*.

Tsekouras, G., Kanellos, F., and Mastorakis, N. (2015). Short term load forecasting in electric power systems with artificial neural networks. In *Computational Problems in Science and Engineering*, pages 19–58. Springer.

United-States-Environmental-Protection-Agency (XXXX). Electricity customers.

Wang, Y., Skerry-Ryan, R., Stanton, D., Wu, Y., Weiss, R. J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., et al. (2017). Tacotron: Towards end-to-end speech syn. *arXiv preprint arXiv:1703.10135*.

Xie, J., Chen, Y., Hong, T., and Laing, T. D. (2018). Relative humidity for load forecasting models. *IEEE Transactions on Smart Grid*, 9(1):191–198.

XLSTAT (2017). Using differencing to obtain a stationary time series.

Zhang, A., Song, S., Wang, J., and Yu, P. S. (2017). Time series data cleaning: from anomaly detection to anomaly repairing. *Proceedings of the VLDB Endowment*, 10(10):1046–1057.