INDUSTRIAL MULTI-STEP TIME SERIES FORECASTING WITH MACHINE LEARNING METHODS

by

Joseph Knutson

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

June 2019

Abstract

Time series forecasting is a powerful tool for predicting the future values of variables involved in industrial processes, identifying their patterns through autoregression. Making forecasting models of industrial processes requires large amounts of industrial data, which is becoming more plentiful due to industrial digitalization. However, companies are aware of their datasets' value and do not necessarily make their data publicly available. Luckily, we have had the fortune of analyzing a utility company's industrial dataset which consists of measurements from thousands of sensors inside of Oslo's district heating system. The goal of our analysis has been to predict local differential pressure, inside a fixed location in the system, up to 40 minutes into the future. Using Artificial Neural Networks, such as LSTMs, it has been shown that they can better predict the differential pressure than models commonly used (as e.g. persistence or linear AR models).

To whom it may concern

Acknowledgements

I would like to offer my gratitude to my supervisor Morten and my co-supervisor Håkon. Morten, your lectures have always been my favourites. Your speech and personality have a warmth to them, as well as an entertaining charm. Håkon, I still remember meeting you when I was a bachelor's student, and you a master's student. Like Morten, you had a welcoming attitude and offered me a place to study among the other master's students. As a result, this office quickly became my home. I want to thank you and Morten, for your hospitality, advise, suggestions and proof reading.

Secondly, I would like to thank Harald, Are, Espen, Bertil, Leif and the rest of Intelecy. Letting me cooperate with you gave me the opportunity to study subjects that I am highly passionate about. Extra thanks to Harald and Are, whose competence exceeds mine on the subjects surrounding this thesis. Your advice and proof reading have been crucial for the completion of this thesis.

The third group that deserves my thanks, perhaps the most important group, is the people that have made every day enjoyable. Emilio, your ability to listen to my rants have been a dear addition to the days I have felt the most frustrated. To the guys at the computational physics program at UiO; owning you in Super Smash Bros. and Mario Cart rarely felt repetitive. The talks we have had together during lunch often span far beyond the realms of physics and have changed my outlook several times. Mom, thank you for always having plate of food ready when I come home at night. To my sister, Emili, thank you for proof reading the entire thesis, while also getting joy out of it. And of course, to the present and former members of the D&D gang; the peak of the week has always been our tabletop nights. Espen, Marius, Stian and Andreas, I hope we get to keep adventuring, even though most of us are transitioning into the workforce now. A new addition to the gang, Irja, also deserves a mention, your quick responses and humorous personality have been welcome additions to my daily motions.

Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

ANN	Artificial Neural Network
Neural Network	Artificial Neural Network
Network	Artificial Neural Network
FFNN	Feed Forward Neural Network
BPNN	Backpropagating Neural Networks
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
LSTM	Long Short-term Memory
SLSTM	Stacked Long Short-term Memory
MLP	Multilayer Perceptron
DMF	Direct Multi-step Forecasting
ML	Machine Learning
Pitts Neuron	McCulloch Pitts Neuron
SLR	Simple Linear Regression
AR	Autoregression
OLS	Ordinary Least Squares
PDT	Differential Pressure Transmitter
DHS	District Heating System
RES	Renewable Energy Sources
PT	Pressure Transmitter
TT	Temperature Transmitter

MSE	Mean Squared Error
MAE	Mean Absolute Error
w.r.t.	With respect to
r.h.s.	Right hand side
l.h.s.	Left hand side

Chapter 1

Introduction

This chapter contains the motivation and goal behind this thesis as well as a description of the thesis' general structure.

1.1 Motivation

The living conditions of a country are normally dependent on its energy usage (MPE 2008). However, human energy consumption is also considered one of the main contributions to the problem of climate change (IPCC 2007). To solve this conundrum, we actively need to transition to more renewable energy sources (RES). The economic consequences of this transition are not necessarily detrimental as studies have shown positive economic effects in multiple countries from the introduction of RES (Apergis and Payne 2010). In the thermal sector, it is known that district heating systems (DHS) can play a pivotal role in the exploitation of RES (EU 2013). DHS, with their low carbon emission levels and cost-effective heat production, bring affordable heat to districts without the need for electricity. DHS provide a notable part of the energy consumption per capita in many countries, such as Finland, Sweden, Denmark and Germany.

Some of the biggest industries today, like the manufacturing and utility industry, monitor more and more of their systems' processes with sensors. District heating systems are no exception. These sensors generate large quantities of historical unprocessed data, which is then processed and saved in a database-like infrastructure (ABB 2013). Taking recent advances in machine learning (ML) into account, we hypothesize that this data has a lot of unrealized value. Many applications can be developed through the use of historical sensor data, such as:

- **Anomaly Detection**

  Also called outlier detection, anomaly detection is a way of detecting anomalies from recent data. One way of performing anomaly detection is to check if a measured value is over or below a certain threshold value.

- **Anomaly Prediction / Predictive Maintenance**

  Accurate forecasting models, usually created with large amounts of historical data, can predict anomalies that have not yet taken place. Knowing the state of the processes and equipment, an organisation can perform a maintenance when it best fits them. This can help reduce costs and downtime, as maintenance sessions usually are held in fixed intervals.

- **Process Optimization**

  Manually adjusting the state of a process in order to optimize it can be time consuming. One can use ML algorithms, trained on vast historical datasets, to automatically find patterns of adjustments that lead to an optimal process.

With these valuable applications in mind and the large quanitity of industrial data, training predictive ML algorithms on historical data from DHS seems like a worthwhile task, as new insights may help with optimizing the use of heating systems.

1.2 Goals

The goals of this thesis are:

- Create a data pipeline which processes historical sensor data from a DHS, turning the data into uniform time series.
- Create artificial neural networks and recurrent neural networks for forecasting pressure sensor data from a DHS.
- Implement a traditional linear autoregression model and a persistence model to use as benchmark models.
- Analyze time-series data well enough to make predictions that are competitive with already established forecasting methods.
- Compare and discuss the performance of the different forecasting models.

1.3 Contributions

In this thesis, we have contributed the following:

- A data pipeline that resamples and forward fills aperiodical data, turning it into uniform time series which are easier to analyze.
- Standard implementations of ANN and LSTM models from the Keras library in Python, as well as some specialized LSTM models (Stacked LSTMs and LSTMs with embeddings).[1]
- A comparative analysis of ANNs and classical forecasting methods.

1.4 Thesis Structure

This thesis consists of five parts; Background, Theory, Methodology, Analysis/Results and Summary. The smallest of the five is the background chapter where we introduce our problem, information around it, as well as briefly touching on the methods we plan on using. Theory, as the name suggests, holds most of the theory regarding regression and optimization methods. In the Methodology part we present our data pipeline and discuss our various heuristics we have made to reduce the amount of irrelevant data in our dataset. The fourth part, Analysis and Results, discuss different hyperparameter settings, architectures and input features and how they affect prediction. We also compare our neural network models with each other and with our benchmark models. In the last part, Summary, we take a critical look at what we done and try to make sense of our analysis results before we conclude. Lastly we discuss our contributions and future perspectives.

[1]The code for both the pipeline and the models can be found on my github at: https://github.com/mathhat/Timeseries-Forecasting

Chapter 2

Background

This chapter consists of basic theory and problem information. We start off explaining what time series are. Secondly, we describe what kind of system our data is generated from before we present our methods.

2.1 Time Series Forecasting

Before we discuss time series forecasting, let us briefly explain what a time series is.

2.1.1 Time Series

Time series data constitutes data that has a natural order in time, such as weather data measurements, stock prices, or a particle's whereabouts in time and space. Imagine a set of timestamps $\mathbf{T} = \{t_0, t_1, t_2, t_3, ..., t_N\}$. A time-series is then the set of observations x_t, where each observation, x, corresponds to the specific timestamp, t, when the observation was measured. The timestamps are typically uniformally spaced in time, meaning there is a fixed time interval between each observation. Industrial historical data, however, are usually generated from event-driven sensors. Event-driven meaning: A measurement that is saved when it is sufficiently different from the previously saved measurement. "Event-driven" will be used multiple times from this point and will always refer to the preceding definition. This means the time series data we will be working with are generally non-uniformly spaced.

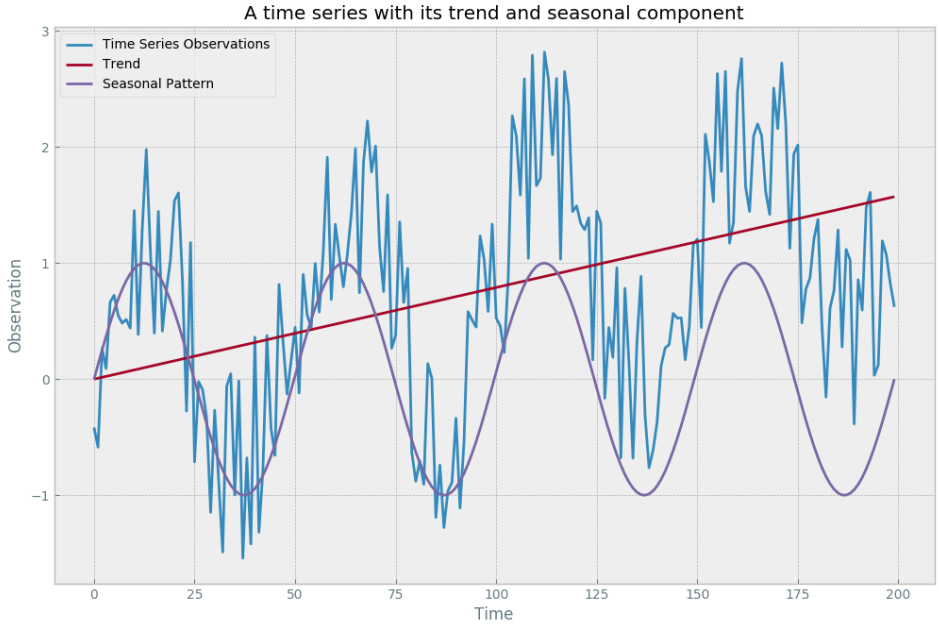

Figure 2.1: An example of a time series with both seasonality and trend. The trend and seasonality have been visualized as two individual graphs.

2.1.2 Forecasting

Time series forecasting is the use of a model to predict future values of a time series, using historical data from the same series (univariate forecasting), or using historical data from several series (multivariate forecasting). A common approach in time series forecasting is to model a series as a combination of an oscillating "zero-mean" function $y(t)$, and a slowly changing trend function $m(t)$ (Brockwell and Davis 2002)

$$\hat{x}_t = m_t + y_t. \tag{2.1}$$

A randomly generated example of such a time series can be seen in figure 2.1.

There are multiple ways of creating models from datasets. In the field of time series forecasting, the methods are separated into two domains: spectral (frequency) domain and temporal (time) domain. Frequency based methods typically apply methods like Fourier and Wavelet transformations to exploit the series' frequencies (Brockwell and Davis 2002). While time based methods prefer to handle the data in its raw form. The way we apply neural networks the forecasting problem is similar to methods from the time domain, as we infer no Fourier or Wavelet transformations on the data.

2.1.3 Multi-step Forecasting

The the type of forecasting we will use in this thesis is called multi-step forecasting. Multi-step forecasting simply means that we will be predicting more than one timesteps into the future. Typically, one only predicts a single timestep into the future. By performing multiple predictions (multi-step forecasting), at different distances into the future, one can get a continuous perspective of the variable's estimated future. The way we choose to predict multiple timesteps into the future is by creating multiple models, one for each timestep into the future. This is known as Direct Multi-step Forecasting (DMF).

2.2 The District Heating System

This thesis revolves around predicting sensor data from a DHS in the form of a third generation heat network. A third gen. heat network transfers heat via pressurized water in insulated pipes, typically buried beneath the ground. Heat networks continuously lose their heat as a consequence of both environmental heat dissipation and customer heat consumption, the latter being what the network is made for. Heat dissipation is a problem for these networks, as 5-10% of the energy the system carries spills into its surroundings.

2.2.1 Infrastructure

Conceptually, a heating system is a heat transfer network as depicted in figure 2.2, with an interwinding network structure that connects a city's buildings together. The heating system our data is from is situated under the city of Oslo, the capital of Norway, and is the largest DHS in Norway. The city's size is approximately 454 square kilometers, a bit less than one-third the size of London which gives an estimate of the maximum dimensions of such networks. The water in the network is heated by multiple boilers, inside heat production plants. The network's size, amount of heat sources and the overlapping grid structure are all factors that add structural complexity.

2.2.2 Energy Sources

The central heat plants are all unique. They can contain multiple boilers, and not just for one type of fuel. Some boilers run on electricity when electricity prices are low, using large heat pumps. More popular than the heat pump boilers are the

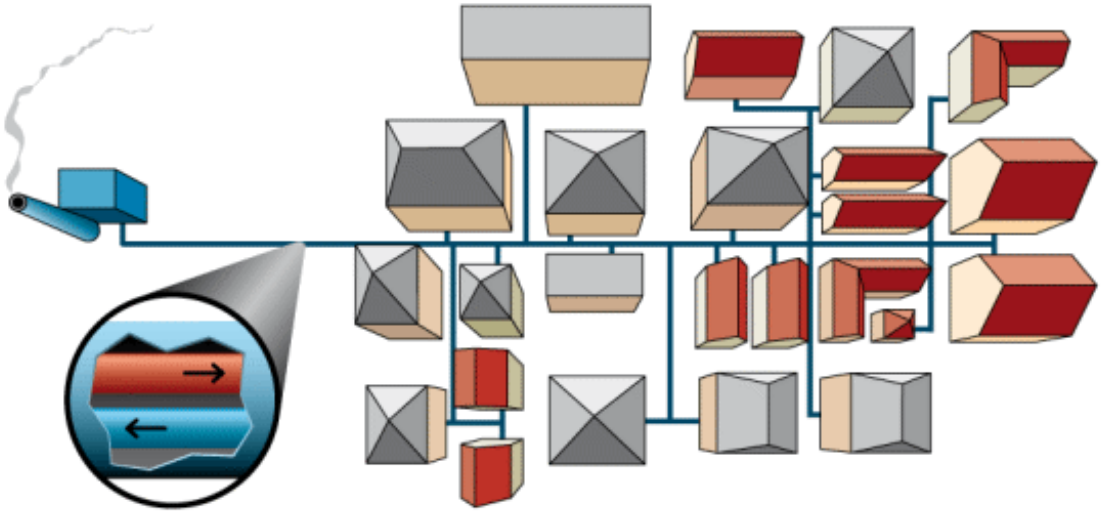

Figure 2.2: A simplified heating system. The magnified circle in the figure illustrates that the water coming out of the heat plant is heated, while the water returning from the households is cooled.

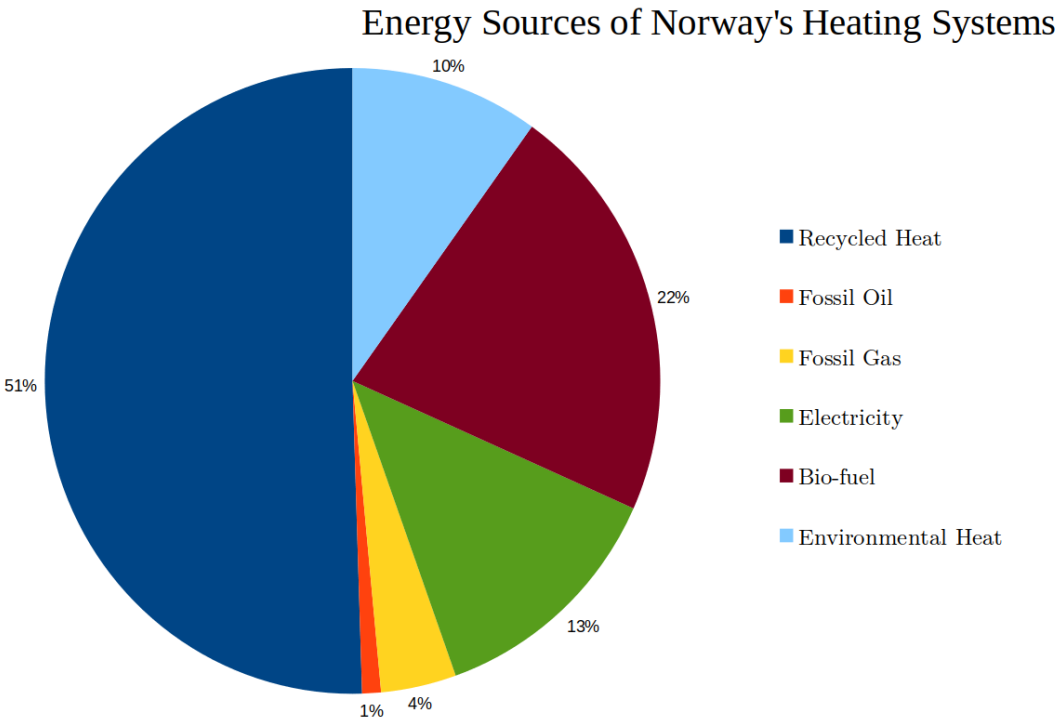

Figure 2.3: Pie chart of the most utilized fuel sources by Norway's heating systems.

biofuel boilers. The largest energy source of all is recycled heat. As of 2017, 51% of all the energy fed to Norway's heating systems came from from recycled sources. The recycled heat comes mainly from burnt waste (92%) and leftover heat from industrial processes (6%). In figure 2.3 is a pie chart illustrating the distribution of the different fuel types that Norwegian heating systems utilize.

The reason why the heat from disposed waste is labeled recycled heat is rationalized like this: Since a natural deterioration of the trash would release CH_4 instead of CO_2 gas, the former being a much more potent greenhouse gas, burning the trash is a viable strategy in combating climate change. In addition to the burning arguably being environmentally friendly, Norway's garbage recycling program ensures that a minimal amount of the waste being burnt comes from a fossil origin. The most valuable benefits from burning the waste is the electricity it generates, but a lot of heat goes unused in this process. The excess heat generated from the electricity production can therefore be recycled into our heating system, which otherwise would have gone to waste.

2.3 Neural Networks for Forecasting

Artificial neural networks (ANNs) are viable tools for the task of forecasting industrial historical data. Although ANNs have a limited use in today's time series forecasting toolbox, their performance is well documented. Already a couple of decades ago it was shown, by Kolarik and Rudorfer (1994), that a simple artificial neural network can outperform well established methods for forecasting time series. This is likely because ANNs are able to model non-linear relationships in the data (Hornik, Stinchcombe, and White 1989), while older time series forecasting methods are typically linear models (Zhang 2012).

2.4 The Data

The historical data from the heating system's thousands of sensors, or "tags", make up the entirety of our dataset. Their measurements are timestamped, cleaned for outliers and saved. The data spans a one and a half year period from 2017-2018 C.E. In this section, we will give some superficial insight into both the sensors and the data we use.

2.4.1 Assets and Tags

Inside the heating system are dynamic assets who regulate and monitor different processes, such as flow regulators and heating ovens. On these assets are sensors, which will be referred to as "tags" from here on. These tags are event-driven sensors, but have a constant sampling rate of 1 Hz, meaning that a tag does a measurement every second and if that measurement is different enough from the last, the measurement is saved. Each tag measures a specific quantity, e.g. temperature, pressure and differential pressure. Some quantities are unitless, meaning they monitor/represent something non-physical, such as a binary value representing an open or closed gate. There are tens of thousand of tags. This makes part of our thesis problem to not only make models, but also to find the most relevant tags focus on. The words; tag, variable, input feature and response variable all refer to the array of measurements of a tag in this thesis.

2.4.2 Format

The historical data is a series of rows, where each row contains the timestamp, the tag name, the measurement, and various other values which we do not have to worry about. Below is a sample of measurements taken from a single tag.

```
"2018-03-09 22:22:00","XXX_PT705.vY",747.59,192,192,4
"2018-03-09 22:23:00","XXX_PT705.vY",733.39,192,192,4
"2018-03-09 22:24:00","XXX_PT705.vY",735.43,192,192,4
"2018-03-09 22:25:00","XXX_PT705.vY",739.50,192,192,4
"2018-03-09 22:26:54","XXX_PT705.vY",766.21,192,192,4
"2018-03-09 22:27:12","XXX_PT705.vY",739.57,192,192,4
"2018-03-09 22:28:01","XXX_PT705.vY",750.72,192,4
"2018-03-09 22:29:01","XXX_PT705.vY",746.80,192,192,4
```

The value the sensor measures is the first number to the right of the tag name. The unit of the value is given by the tag's name, XXX_PT705.vY, which contains the initials "PT", or, "pressure transmitter". The initials are part of an international standard called the ISA standard/codes(*ISA Codes* n.d.), which is a naming convention for process intstrumentation. The first letter in an ISA code(e.g. P in PT), defines the unit of the measurement. Below is a list of fictional examples of sensor names and what they measure:

```
NYC_TT705.vY, temperature, [C]
LON_PT101.vY, pressure, [kPa]
TOK_FT803.vy, water flow/flux, [kgm^3/s]
PAR_PDT803.vy, differential pressure, [kPa]
```

As you might notice, PT and PDT are different in the sense that PDT measures differential pressure. Differential pressure is simply the difference in pressure at some junction in the heating system. This is the type of sensors we will be analyzing in this thesis.

The last thing to note about the tag names are their first three letters. The first three letters in a tag's name gives away its position, e.g. NYC = New York City. The utility company who provided us with the data has asked us to withhold this information, which is why all locations in this thesis will be referred to as "XXX". XXX is always an arbitrary district within the state/city of Oslo.

2.4.3 Aperiodicity and Asynchronicity

Aperiodicity

In the listed measurements in subsection 2.4.2, it would seem the tag makes a measurement every minute, but the fifth listed measurement breaks with this pattern. Even though the tags have a measurement frequency of 1 Hz, which is periodic, the data is saved aperiodically due to the sensors being event driven. An event driven sensor only saves a measurement when a substantial enough change happens, usually a fixed percentage of the value's expected range. This means our data is aperiodical, and difficult to work with unless we resample it to a periodic frequency. If the sensor had saved every single value it measured, it would generate tens of times more data, which can be cumbersome to deal with.

Asynchronicity

The tags in Oslo's DHS are numerous and mostly independent of each other. A consequence of this is that one tag's measurements usually happen at completely different times than another one. We call this asynchronicity, and it makes multivariate forecasting difficult. The models we intend to use are not designed to work on aperiodical time series who are asynchronous with each other. Our solution to this is in our data pipeline, where we make the time series uniform/periodical and synchronized.

2.4.4 Weather Data

We hypothesize that our prediction problem is easier to solve if we bring data from outside of the heating system. This is because human behavior, such as Oslo's

citizens' tendency to take showers, may have an impact on the heating system. Perhaps weather affects showering patterns? Data we give our models from outside of the heating system can be categorical, such as telling our model which day it is, or numerical, like telling the model how windy it is. Our weather dataset contains 10 minute interval measurements of wind direction, wind strength, max gust, long term precipitation and 10 minute precipitation:

```
timestamp         w_d  w_s max_w tot_p   p_10
01.01.2017-00:00  40   0.6  1.2   343.1   0
01.01.2017-00:10  37   1.0  1.4   343.1   0
01.01.2017-00:20  71   0.7  1.2   343.1   0
01.01.2017-00:30  13   0.5  1.4   343.1   0
01.01.2017-00:40  14   0.4  1.0   343.1   0
01.01.2017-00:50  72   1.1  2.1   343.1   0
```

The weather data comes from the Norwegian Meteorological Insititute's weather station at Blindern, Oslo, Norway and is free for anyone who would wish to analyze it.

Part I

Theory

Chapter 3

Supervised Machine Learning

The point of supervised learning is to create a model which learns to map an input to an output based on input-output pairs it has seen before. In this chapter we will discuss some basics of supervised learning, while mainly focusing on the part of supervised learning called regression.

3.1 Regression or Classification

Supervised learning can be partitioned into two sub classes, classification and regression. What separates classification and regression in supervised learning is simply the output of the input-output pairs. The input in both cases is usually a vector which is mapped to an output signal that is attempting to mimic the output of the input-output pair, the label. If you are training a classification model, the labels are categorical, e.g. “green”, “dog”, “False”, or “True”. The model, once it has learned a pattern between the input-label pairs, can then be used as a classifier for inputs that not yet possesses a label/class. In the case of regression models, the labels they learn to mimic are real-valued, e.g. 42.69cm or 13.37s. Classification is a field in statistics that deals with categorizing datapoints into their corresponding classes, while regression is a field in statistics that tries to model the relationship between the input and label. The two methods are illustrated in figure 3.1. Of the two, it is regression that this thesis revolves around.

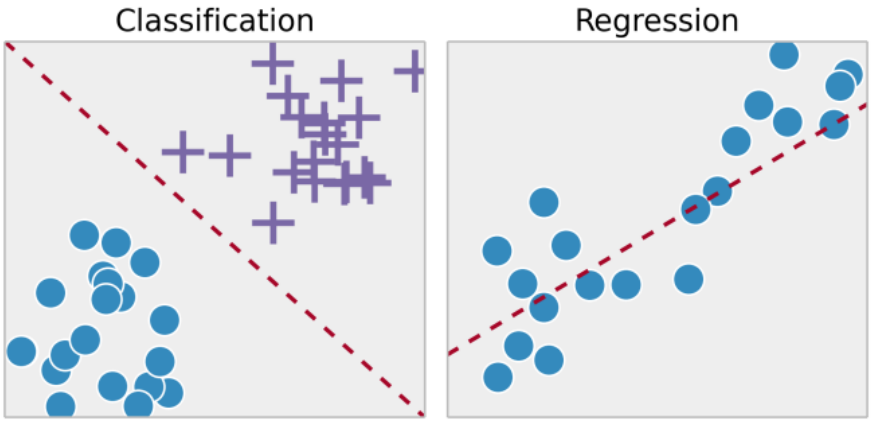

Figure 3.1: Supervised learning consists of both classification and regression. Left: An example of a classifier that is able to distinguish between two types of datapoints. Right: A regression model that has found a trend in the data. Image source: Edell (2015).

3.2 Linear Regression

Regression is the statistical measure of estimating the relationship between variables. If you for example have a set of discrete temperature measurements $\mathbf{T} = \{T_0, ..., T_n\}$ and the times of day these measurements were taken $\mathbf{t} = \{t_0, ..., t_n\}$, you can perform a regression analysis to approximate the relationship between temperature and time of day; $\hat{\mathbf{T}}$. Just like supervised learning, this is synonymous with teaching a model to find the relationship between the input-output pairs in a dataset. In figure 3.2 is a visualization of a model such as $\hat{\mathbf{T}}$. The purpose of this section is to present the basics of linear regression and show how one goes about to optimize a model.

Regression has two general purposes:

- **Correlation analysis**: A correlation analysis finds out whether or not two variables are correlated. In this thesis, we check for both linear and non-linear correlation within our data pipeline.

- **Regression analysis**: As expressed in Zou, Tuncali, and Silverman (2003), in a regression analysis, the variable we want to predict the behavior of (label) is called the response variable, while the model's input variable is called the explanatory variable. Regression analysis is to model and predict a response variable (e.g. temperature) as a function of another variable (e.g. time).

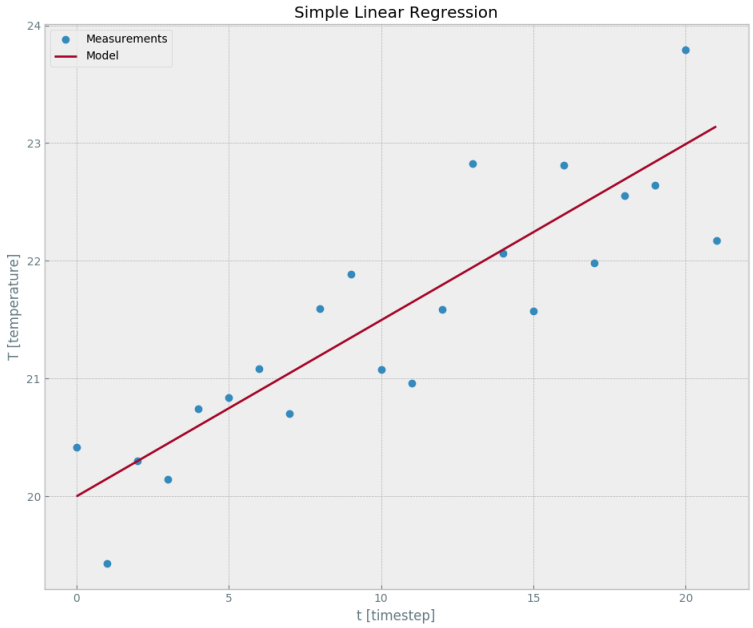

Figure 3.2: A linear model of our temperature measurements. The data is artificial.

3.2.1 Simple Linear Regression

Simple linear regression analysis assumes that the relationship between your variables, e.g. temperatures $\mathbf{T}$ (outcome variable) and timestamps $\mathbf{t}$ (predictor variable), is linear. The general goal of SLR is to draw a straight line as close to all your datapoints as possible (see right-hand side of figure 3.2). This line is expressed in eq. (3.1)

$$\hat{T}(t) = \beta t + \beta_0, \tag{3.1}$$

where $\hat{\mathbf{T}}$ are the estimated temperatures, $\mathbf{t}$ are the measured timestamps, β denotes the trend in the model and β_0 is the line's intercept. The parameters β and β_0 in eq. (3.1) must be optimized (tweaked) to ensure a good fit between the line $\hat{T}$(t) and the measurements T(t).

3.2.2 Ordinary Least Squares Optimization

The optimization of a regression model is synonymous with the minimization of its cost function (see section 4.7.2 for more information regarding cost functions). In order to optimize a model $\hat{\mathbf{T}}$, we first need to define a function that expresses how wrong it is. Such a function is called a cost function, and typically accumulates

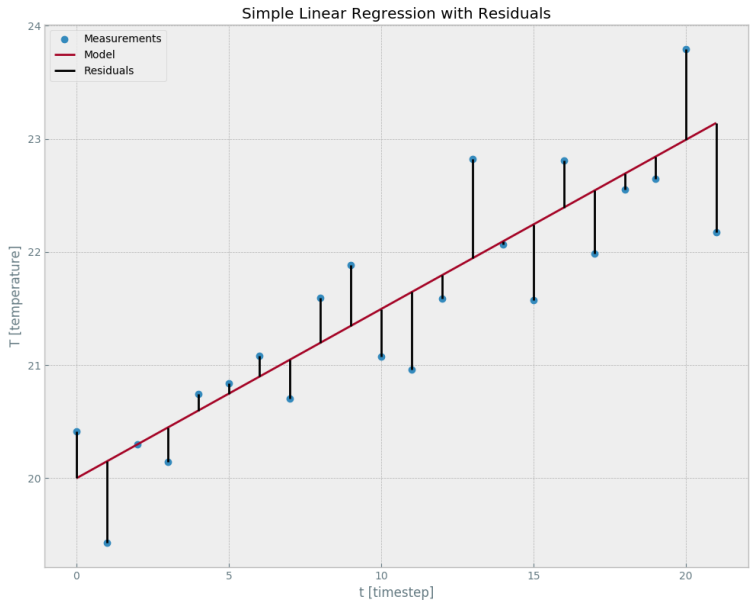

Figure 3.3: The same model visualized in figure 3.2, except that we have now visualized the residuals, the errors of the model. The point of regression is to minimize the sum of all these residuals.

all the errors the model has made. We can define the errors of our hypothetical temperature model as the differences between the predicted temperatures $\hat{\mathbf{T}}$ and the temperature labels $\mathbf{T}$. These errors are named *residuals*, or r_i

$$\begin{aligned} r_i &= T(t_i) - \hat{T}(t_i) \\ &= T(t_i) - \beta t_i + \beta_0. \end{aligned}$$

In figure 3.3 we display the residuals between the linear model and the measured labels from figure 3.2. To express all the errors as a function, we sum up all the residuals. When performing OLS regression, we square the residuals before summing them. This is because some residuals are likely to be negative, and others positive, meaning the residuals would simply cancel each other out if we summed them, giving us a smaller error than what is true. Squaring and summing each residual gives us

$$\begin{aligned} S_{ssr}(\beta, \beta_0) &= \sum_{i=1}^{n} r(t_i)^2 \\ &= \sum_{i=1}^{n} (T(t_i) - \beta t_i + \beta_0)^2. \end{aligned} \tag{3.2}$$

The function S_{ssr} in eq. (3.2) is known as the **sum of squared residuals**.

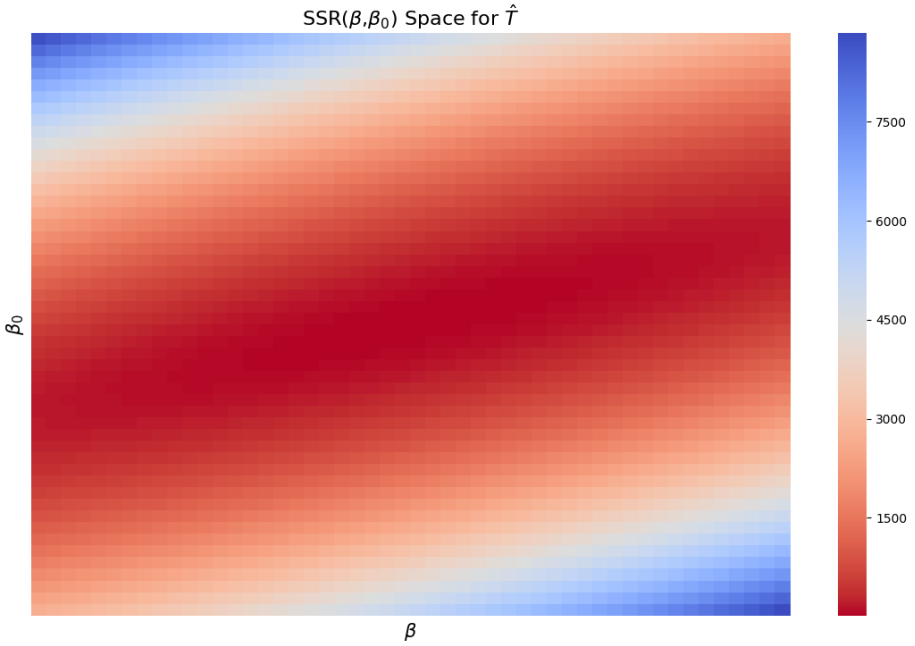

Figure 3.4: A heatmap of the SSR for various temperature models, such as the one in figure 3.3. The optimal values for β and β_0 seems to be positioned in a diagonal valley across the figure. This is where the residuals are the smallest and yields the best fitting line through our data.

3.2.3 Minimizing the Cost Function

In figure 3.4 is a plot of how the error in figure 3.3 behaves as a function of β and β_0. It is evident that reducing the error S_{ssr} is done by finding the optimal parameter values β and β_0 for our model. To begin finding the optimal parameter for models, we need to derive the gradients of their cost function. Before differentiating the cost function expression w.r.t. β and β_0, we need to expand it

$$\begin{aligned} S_{ssr}(\beta, \beta_0) &= \sum_{i=1}^{n} (T(t_i) - \hat{T}(t_i))^2 \\ &= \sum_{i=1}^{n} (T(t_i) - (\beta t_i + \beta_0))^2 = \sum_{i=1}^{n} (T(t_i) - \beta t_i - \beta_0)^2 \\ &= \sum_{i=1}^{n} [T(t_i)^2 - 2\beta_0 T(t_i) - 2\beta t_i T(t_i) + \beta_0^2 + 2\beta_0 \beta t_i + \beta^2 t^2] \\ &= \sum_{i=1}^{n} T(t_i)^2 - 2\beta_0 \sum_{i=1}^{n} T(t_i) - 2\beta \sum_{i=1}^{n} t_i T(t_i) + n\beta_0^2 + 2\beta_0 \beta \sum_{i=1}^{n} t_i + \beta^2 \sum_{i=1}^{n} t_i^2. \end{aligned} \tag{3.3}$$

Gradients of the SSR

Below are the partial derivatives of a model's SSR with respect to both its parameters, which we for practical reasons call "gradients"

$$\begin{aligned}\frac{\partial S_{ssr}}{\partial \beta}(\beta, \beta_0) &= -2\sum_{i=1}^{n} t_i T(t_i) + 2\beta_0 \sum_{i=1}^{n} t_i + 2\beta \sum_{i=1}^{n} t_i^2 \\ &= -2\sum_{i=1}^{n} \left(t_i T(t_i) - \beta_0 t_i - \beta t_i^2 \right), \end{aligned} \tag{3.4}$$

$$\begin{aligned}\frac{\partial S_{ssr}}{\partial \beta_0}(\beta, \beta_0) &= -2\sum_{i=1}^{n} T(t_i) + 2n\beta_0 + 2\beta \sum_{i=1}^{n} t_i \\ &= -2\sum_{i=1}^{n} \left(T(t_i) - \beta_0 - \beta t_i \right). \end{aligned} \tag{3.5}$$

Exploiting these error gradients can be done either analytically or numerically. Minimizing a cost function numerically is usually done in a gradient descent scheme (see sector 4.7.4 for more info on numerical optimization). In a gradient descent scheme we look at the sign of the gradients in order to know which direction to tweak our parameters in order to minimize the error. These tweaks usually happen for a fixed number of iterations.

The analytical way to minimize the error would be to set the r.h.s. of the gradients' expression equal to zero and find a closed form solution for each of our parameters. Few models have closed form solutions to their gradient expressions and so numerical optimization is required for most supervised learning models. The analytical solution for OLS does exist and is much more efficient than a numerical optimization scheme. If the reader is interested, we have derived the analytical solution to OLS regression in appendix (A.1).

3.3 Fit and Predict

In **regression analysis**, most methods rely on fitting a model $\hat{y}(x)$ on a dataset ($\mathbf{x}$,$\mathbf{y}$), so that some error function, e.g. the sum of squared residuals $\sum_i (y_i - \hat{y}_i)^2$, is minimized. After fitting, one can either analyze how accurate $\hat{y}$ models y, or one can attempt to infer the model $\hat{y}$ on new data observations $(\boldsymbol{x}_{test}, \boldsymbol{y}_{test})$ in order to see how well it predicts the new observations (**prediction**). In this section we take a closer look at regression analysis and the process of fitting and predicting.

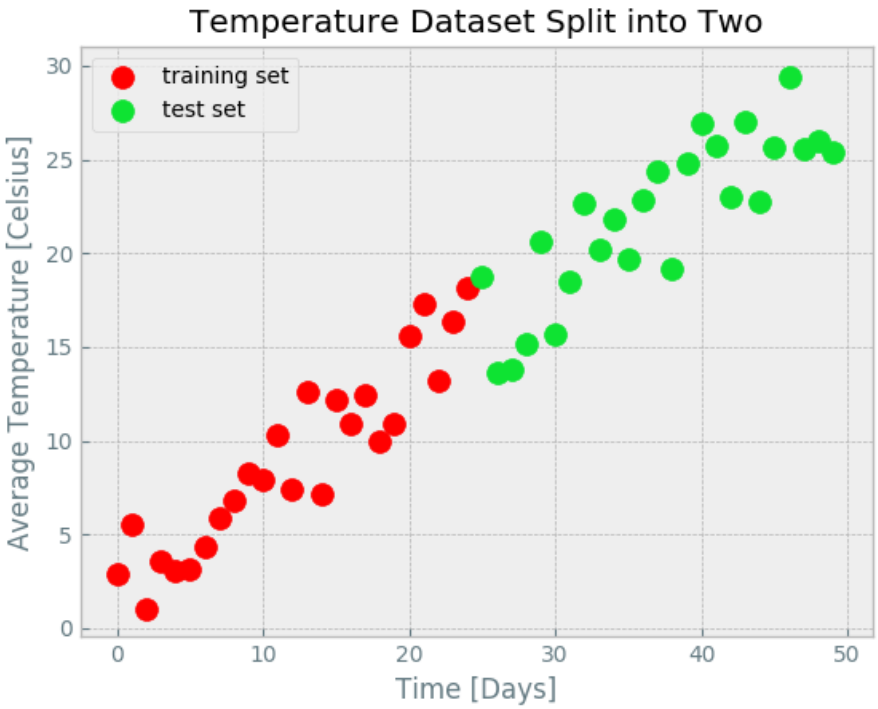

Figure 3.5: The figure depicts an artificial dataset portraying 50 measurements taken over the course of 50 days. In red is the part of the dataset one should use to make a model of the temperature. In green is the part of the data one should not train on, but rather test their model with.

3.3.1 Fitting

The data used to fit, or train, model $\hat{y}$ is called a training set. Datasets which we use to evaluate our model are called test sets, and contain datapoints that our model has not been fitted on. The reason why we evaluate, or test, the model on a test set rather than the training set is because our model might have overfitted on the training set during the fitting process (see section 3.4).

As we did in the section on linear regression, let us make another linear model in order to portray the difference between fitting and predicting. First of all, let us randomly generate a dataset $(\boldsymbol{x}, \boldsymbol{y})$ which we split into a training and testset: $(\boldsymbol{x}_{train}, \boldsymbol{y}_{train})$ $(\boldsymbol{x}_{test}, \boldsymbol{y}_{test})$. For practicality, let us pretend $\boldsymbol{x}$ is time and $\boldsymbol{y}$ is a set of observations of daily average temperature. See figure 3.5 for the randomly generated data.

It is common to use 50% of the dataset for training and the other 50% for testing when analysing small datasets, but the bigger the dataset gets, the smaller percentage of the dataset is used for testing. The reason why only a small portion

of large datasets are used for testing is because only so many tests are needed in order to prove the general accuracy of your model. We have for this reason decided to split the dataset in figure 3.5 into two equally sized subsets.

Before fitting, it is normal to shuffle the dataset so that the input-output pairs the model trains on are of a wide variety. When the data is chronological however, one never shuffles the dataset before splitting it into a training and test set. The reason why is because time series contain inherent temporal components which we need to exploit if we wish to model the series accurately. To exploit the temporal connections in the data, the models we make need to be fitted on contiguous parts of the dataset, or else we destroy the "locality" of the data (see section 5.1 for more on locality). This is why the training set and test set in figure 3.5 get to keep their chronological order, making them two contiguous series of data.

Let us fit a simple linear model onto our example training set (the red dots in figure 3.5). As we discussed in section 3.2.3, fitting/minimizing can be done analytically or numerically. Using Python code imported from the scikit-learn package, the implementation we have used performs OLS regression. Remembering that OLS minimizes the **sum of squared residuals**, $\boldsymbol{S_{ssr}} = \sum_i (y_i - \hat{y})^2$, it feels important to note that this minimization will only happen on the training set, $(\boldsymbol{x}_{train}, \boldsymbol{y}_{train})$, when **fitting**.

The minimization of the error $\boldsymbol{S_{ssr}}(\boldsymbol{y}_{train}, \hat{\boldsymbol{y}})$ is known as fitting. This **sum of squared residuals** over the training set can be written as:

$$\boldsymbol{S_{ssr\ train}} = \sum_{x \in x_{train}} (y(x) - \hat{y}(x))^2, \tag{3.6}$$

and the minimization problem can be written as

$$\min_{\hat{\beta}, \hat{\beta_0}} \boldsymbol{S_{ssr\ train}}(\hat{\beta}, \hat{\beta}_0),$$

where $\hat{y} = \hat{\beta}x + \hat{\beta}_0$. The fitted model $\hat{y}$ is seen in figure 3.6.

3.3.2 Predicting

Having now fitted a linear model on the training set, we want to see how well it fits new data / test data. Our model has only been trained to model the average temperature of the first 25 days of the dataset, so how does one go about to predict temperatures at a later date? The answer is: By simply inserting x values (or dates) belonging to the test dataset, $\boldsymbol{x}_{test}$, into $\hat{y}$. Doing this extends our model into unseen territory (see figure 3.7).

To measure how well our model predicted the future temperature measurements in the test set, we compute the residuals between our model $\hat{\boldsymbol{y}}(\boldsymbol{x}_{test})$ and the actual

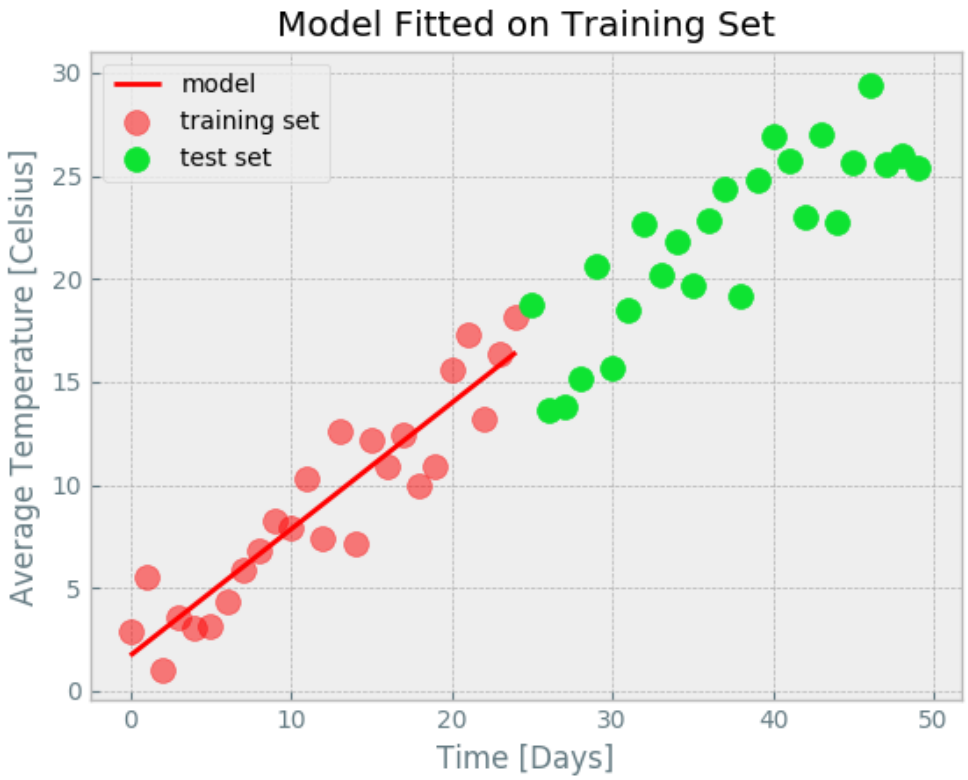

Figure 3.6: Using Ordinary Least Squares (OLS) we have fitted a simple linear regression model to the training set.

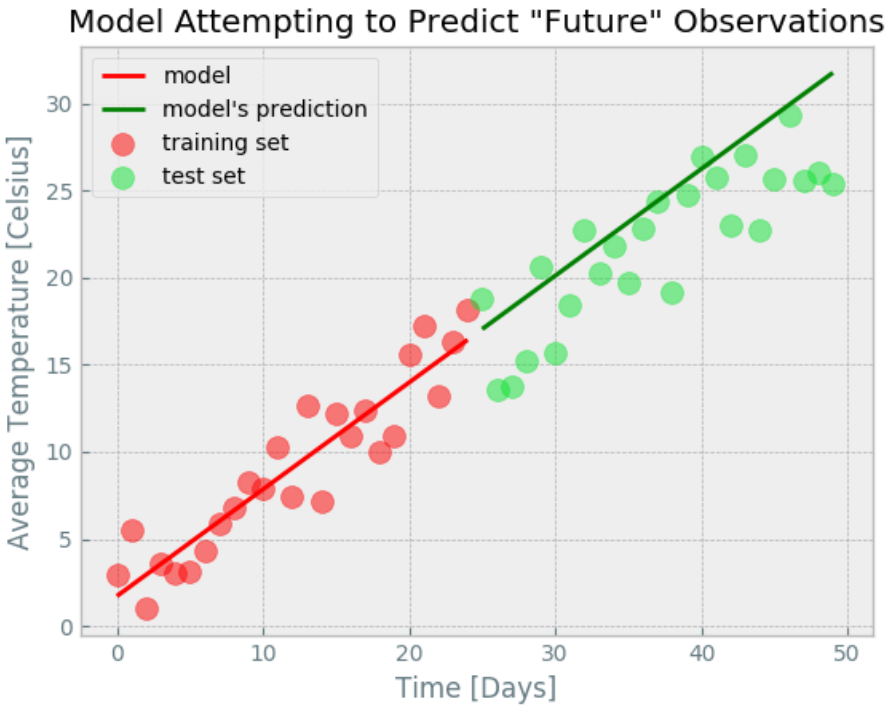

Figure 3.7: This graph shows the same model from figure 3.6. All we have done is to infer it on future dates: $\hat{\boldsymbol{y}}(\boldsymbol{x}_{test})$. By doing this, we have predicted the future temperatures. We see that our predictions (green line), on average, are slightly higher than the true observations (green dots).

future temperatures $\boldsymbol{y}_{test}$ (see eq. (3.7)). By "future" we simply mean a time past the training dataset measurements

$$\boldsymbol{S}_{\boldsymbol{ssr\ test}} = \sum_{x \in x_{test}} (y(x) - \hat{y}(x))^2. \tag{3.7}$$

The summed squared residuals from our predictions, from here on called the "test error", $\boldsymbol{S}_{\boldsymbol{ssr\ test}}$, is likely larger than $\boldsymbol{S}_{\boldsymbol{ssr\ train}}$, or "the training error". This makes sense, since the training error is what we minimized when fitting our model. We only care about the test error and want it to be as small as possible, since the purpose of our model is to predict unseen data. A small test error means that our model has is successful at predicting the data pattern, in this example case, the average daily temperature.

3.4 Overfitting

Since we are on the subject of fitting, overfitting seems like a fitting phenomenon to fit in here. Overfitting is a phenomenon that happens when a model is complex and is fitted "too well" onto a training set. To decrease the risk of overfitting models, a basic understanding of prediction bias- and variance is required. The following subsections are heavily inspired by Mehta et al. (2018)'s introduction to the bias-variance tradeoff.

3.4.1 Model Complexity - Polynomial Regression

An intuitive way of discussing overfitting is by looking at complex models. By complex, we mean that a model has many parameters. Polynomials are a great example of models with various complexity. As a polynomial's degrees increase so does its complexity. In figure 3.8 we have fitted two polynomial functions, one of third and one of 14th degree, onto a noisy training set. In the figure we see that the 14th degree polynomial is much more adept at mimicing the training data's pattern, while the third degree polynomial follows the datapoints in a very "generalized" way. However, once we attempt to extend these models onto the test set, the roles are changed. In figure 3.9 we have illustrated the predictions of the models on the test set, where we observe that the third degree polynomial is far more accurate than its more complex cousin. The reason why the more complex polynomial model failed is due to overfitting on the training set. This does not mean that complexity is bad, but we need a way to balance it. To do this, we look at a model's variance.

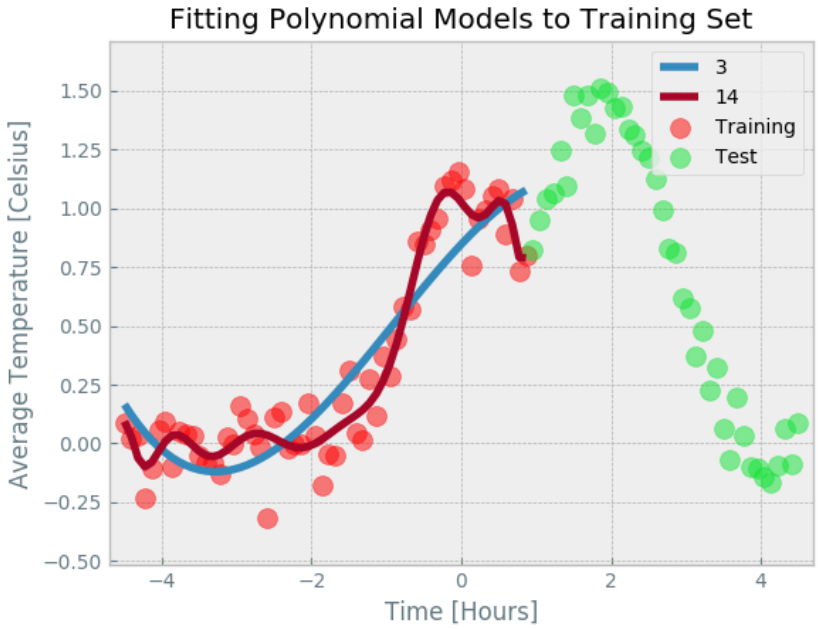

Figure 3.8: Two polynomials of differing complexity have been fitted onto a training set (red). Notice how much more adept the more complex model is at modeling the data's noise.

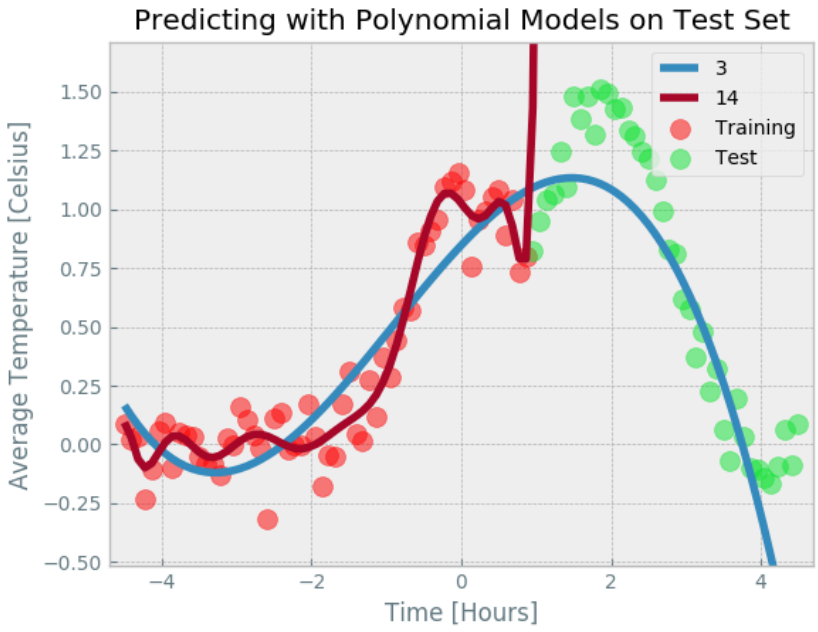

Figure 3.9: Two polynomials of differing complexity attempting to predict the behavior of the test set. Notice the way the less complex model is able to capture the data's general pattern.

3.4.2 Bias and Variance

Given a dataset of N inputs, $\mathbf{X} =\in (x_1, ..., x_N)$, variance is a measure of how much a model, $\hat{f}(\mathbf{X})$ fluctuates around its mean

$$\texttt{Var}(\hat{f}) = \left(\frac{1}{N}\sum_{i}^{N}\hat{f}(x_i)^2\right) - \bar{\hat{f}}(\mathbf{X})^2, \tag{3.8}$$

where $\bar{\hat{f}}(\mathbf{X})$ is the average value of the model.

If a model is very complex, chances are it has a high variance. These models (e.g. neural networks or large polynomials) come with a tendency to fit their training dataset very well, as we saw in figure 3.9, this is not always a good thing. Models rich in variance risk modeling the data they are trained on too well. What we mean by this is that real data contains noise, which we do not want to model. Noise causes data points to deviate, to some degree, from their underlying pattern. High variance models are so sensitive to noise that they end up modeling the nonexistent pattern in the noise instead of the underlying data pattern. This modeling of both the data pattern and the noise pattern is called overfitting and renders the model unable to make any predictions of value.

As we decrease a model's variance, its bias starts increasing. Bias is the difference between the average prediction of a model and the output label it is attempting to predict

$$\texttt{Bias}(\hat{f}) = \bar{\hat{f}}(\mathbf{X}) - y, \tag{3.9}$$

where y is the output label. Models with high bias are relatively bad at fitting their training data and have very little flexibility. An example of a model with high bias is a linear regression model. High bias models do not risk overfitting, but due to their limitations they risk underfitting. Underfitting is a term used to describe when a model fails at picking up a pattern in the data, generally leading to a high error.

The average test error, L, of a model is dependent on both its variance and its bias

$$L(\hat{f}(\mathbf{X}, \mathbf{y})) = (\texttt{Bias}(\hat{f}))^2 - \texttt{Var}(\hat{f}) + \sigma^2 \tag{3.10}$$

where σ^2 is an irreducible. Eq. 3.10 expresses the bias–variance tradeoff which is the problem of how to balance a model's complexity in order to receive the smallest error possible. In figure 3.10 is a visualization of how the tradeoff behaves as a model's complexity increases/decreases. We can demonstrate the bias–variance tradeoff experimentally by creating a lot of polynomial models of varying degrees. In figure 3.11 we have done just that.

There are many different techniques for avoiding overfitting. One can pick a high bias model and not risk overfitting at all. Alternatively, one can pick a complex

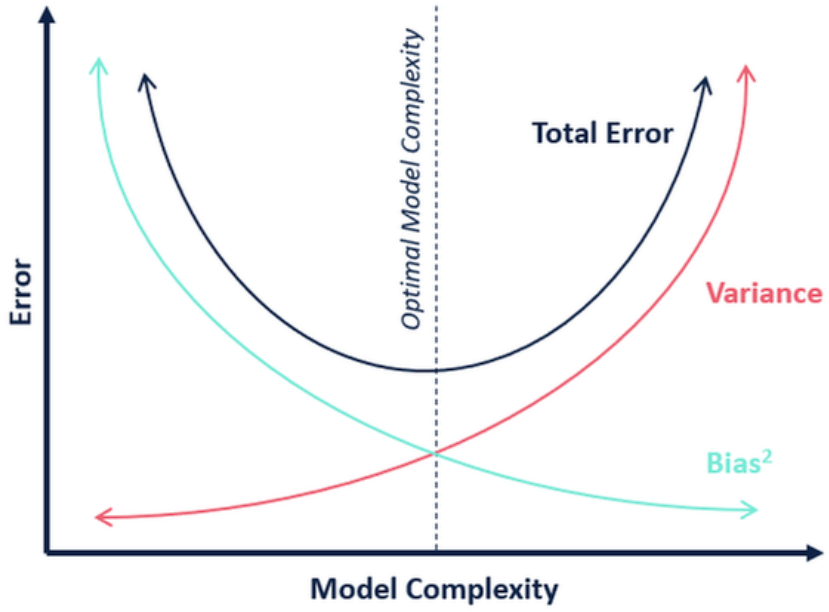

Figure 3.10: The bias–variance tradeoff visualized. The figure illustrates how the error (black) can both decrease and increase as we tweak on a model's complexity.

model and stop training it before it overfits, using a validation set (see next section). A third way to avoid overfitting is regularization, a limitation which helps the model generalize if it is prone to overfit (see section 5.5 for more on regularization).

3.5 Autoregression

As we briefly stated in the introduction, time series forecasting follows a special regression scheme called autoregression. In order to understand the difference between classical regression and autoregression, let us first recapitulate what a standard regression model does in order to predict the future value of a variable. Standard regression approximates the relationship between an explanatory variable (e.g. time) and a response variable (e.g. pressure)

$$\hat{p}_{t+1} = f(t+1) \tag{3.11}$$
$$= \beta(t+1) + \beta_0. \tag{3.12}$$

In eq. (3.12), $\hat{p}_{t+1}$ is the prediction of a future pressure value, where $f(t)$ is the linear regression model that models the pressure w.r.t. time. In time series forecasting, however, it is popular to only use the response variable when creating regression models. To predict future pressure measurements, p_{t+1}, the model is given previous

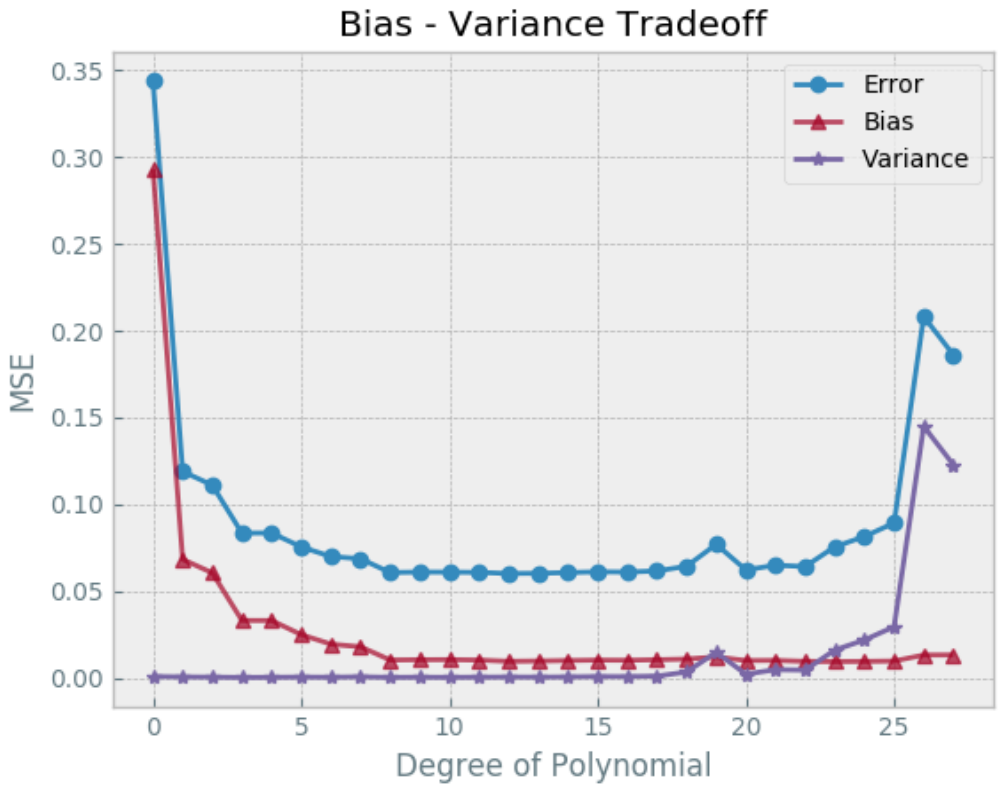

Figure 3.11: The bias–variance tradeoff visualized for some polynomials of varying complexity. The error is the test error, in a similar format as the sum of residuals squared, called **MSE** (see section 4.7.2 for more on cost functions). Notice how the best polynomials are those between 7th and 17th degree. The code to make these polynomials was taken from Hjorth-Jensen (2018).

pressure values:

$$\begin{aligned}\hat{p}_{t+1} &= f(p_t, p_{t-1}, p_{t-2}, ...) \\ &= \beta_1 p_t + \beta_2 p_{t-1} + \beta_3 p_{t-2} + ...\end{aligned} \tag{3.13}$$

This way of performing regression/prediction is called autoregression (AR). Autoregression models are a traditional class of linear forecasting models. In this thesis, we have as goal to implement linear AR models and see how their predictions compare to artificial neural network models.

Chapter 4

Introduction to Artificial Nerual Networks

The first thing that comes to one's mind when machine learning is mentioned might be artificial neural networks. These networks dominate the sub-domains of machine learning called *supervised learning* and *reinforcement learning*, while also being central to the third, *unsupervised learning* (Graves and Clancy 2019). The first notable attempt at creating computational models of biological neural networks started with modeling the neuron. This chapter aims to clarify what an artificial neural network is, how to train it and how to infer it. First however, we present the basic functions of a neuron as well as the artificial neuron of McCulloch and Pitts.

4.1 Neurons

Neurons are specialized brain cells who receive, process and transmit signals, electically and chemically. A general neuron consists of a cell body (soma) and filaments in the form of dendrites and an axon (see figure 4.1). Dendrites are thin filaments that branch out into tree like shapes, some hundred micrometers tall. The Neuron receives electrical signals from other neurons via these dendrites, while outgoing signals are sent via the axon. The axon is a large filament covered in an insulating sheath of myelin, and is similar to a cable. Axons branch out, making multiple connections to multiple other neurons' dendrites. These connections are called "synapses" can regulate all signal flow. Repeating the recursive pattern of the dendrites and axon results in a neural network (see figure 4.2).

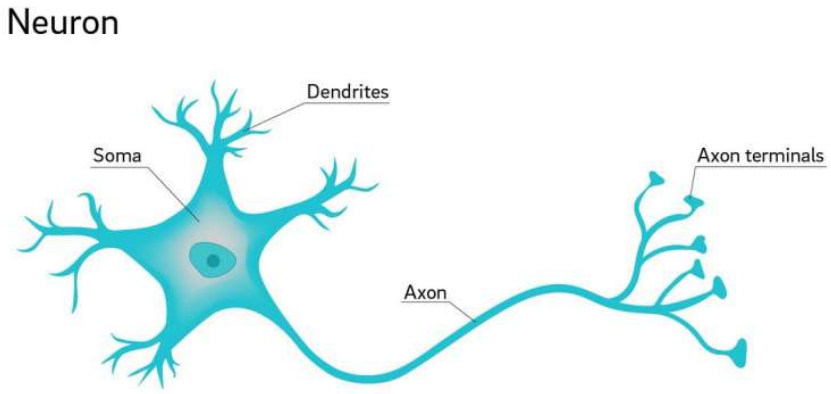

Figure 4.1: A typical neuron has a tiny compact body, the soma. From the soma grows tree-like filaments called dendrite, responsible for receiving signals. The cable like axon is responsible for outgoing signals by connecting to other neurons' dendrite via the axon terminals. *Credits: David Baillot/UC San Diego*

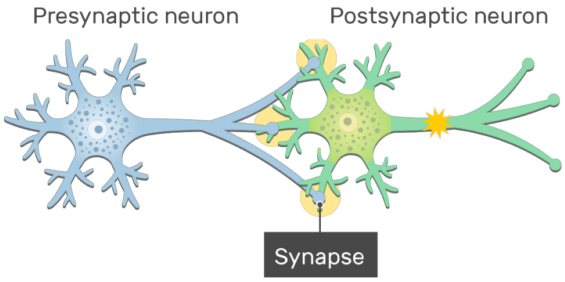

Figure 4.2: Neurons connected via the left neuron's axon and the right neuron's dendrites. The synapse is the junction where the axon of one neuron meets the dendrite of another. Electrical signals in the form of ions usually flow from the axon of one neuron, into the dendrite of another.

4.2 McCulloch and Pitts' Neuron

The predecessor of modern neural networks is the McCulloch and Pitts neuron (McCulloch and Pitts 1943), a simple logical model of the biological neuron. We will sometimes refer to this model as the Pitts neuron for practicality. The Pitts neuron models the biological neuron's ability to become electrically excited by incoming ions. As incoming ions from other neurons react with the synapses along a neuron's dendrite, the ions' interaction with the synapses create a change in the voltage gradient between the incoming synapse and the outgoing synapses. If this change in electrical potential is large enough, it creates an electrochemical pulse called the "action potential". This potential activates the outgoing synapses of the neuron, either causing them to excite or inhibit the next neuron down the line.

For a computational and biological neuron alike, excitation happens when their input signals are sufficiently strong. Additionally, these signals must also interact with the receiving neuron's synapses who can vary in their conductictivity. Taking these two points into account, McCulloch and Pitts introduced three classes of variables to their model: inputs, weights and threshold.

The Pitts neuron consists of n inputs (signals) $x_1,\ x_2,\ ..., +x_n$, who interact with n unique weights (synapses) $w_1,\ w_2,\ ..., +w_n$. We model this interaction as the product between the "signal", x and the "synapse", w. The sum of these variables' products, described in eq. (4.1), is then compared to a treshold value. If the sum is larger than this arbitrary threshold, the neuron fires

$$h = \sum_{i=1}^{n} x_i w_i. \tag{4.1}$$

4.2.1 Threshold Activation

When the neuron fires, it outputs the value "1", if not (due to h being too low), it returns "0". We call the firing for an *activation*. The activation of a Pitts neuron is described with the function

$$f(x, w) = \begin{cases} 0 & \text{if } h < t \\ 1 & \text{if } h >= t, \end{cases}$$

where h is the sum in eq. (4.1) and t is the chosen threshold value. You can see the Pitts neuron visualized in figure 4.3.

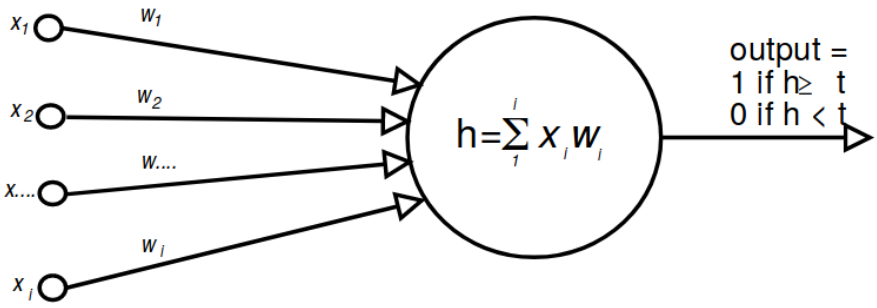

Figure 4.3: An illustration of the McCulloch Pitts neuron with an arbitrary amount of input arguments and weights.

4.3 Feed Forward Neural Networks

Most basic neural networks propogate their nodes' values forward in the network, just like the Pitts neuron. This non-cyclic behavior has dubbed these networks as "feed forward neural networks" (FFNN). Feed forward neural networks are almost identical to Pitts neurons stacked next to each other. A simple example of one is the multilayer perceptron (MLP) (see figure 4.4). A simple MLP consists of input, hidden and output nodes. These three groups of nodes are organized into layers, which you can see in figure 4.4.

4.4 Forward Propagation

Like the Pitts neuron, a hidden node h in an FFNN is a weighted sum of the nodes in the layer before it. When a node is the sum of all the nodes in the layer before it, which is usually the case, we call the layers "fully connected". The mathematical expression for each hidden node then becomes

$$h_j = \sum_{i=1}(x_i w_{ij}) + \beta_j. \tag{4.2}$$

Where h_j is the value of the jth hidden node and x_i is the ith input node. w_{ij} is the weight that connects the jth hidden node with the ith input node and β_j is the bias node connected to the jth hidden node. The bias node gives the network another degree of freedom, which it can use to better fit the data it trains on, kind of like the intercept of a linear regression model.

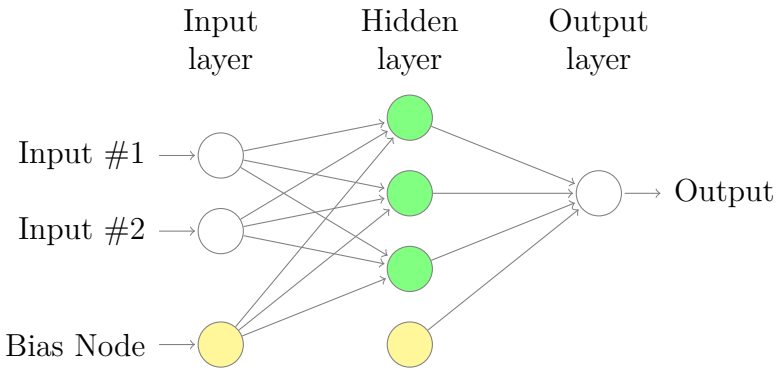

Figure 4.4: A multilayer perceptron with one hidden layer. Each hidden node is a weighted sum of all the previous layer's nodes. The output node is also a weighted sum of its previous layer's nodes. The weights are illustrated as arrows, just like in figure 4.3. The yellow nodes are what we call bias, which play a similar roll in neural nets as intercepts do in linear functions.

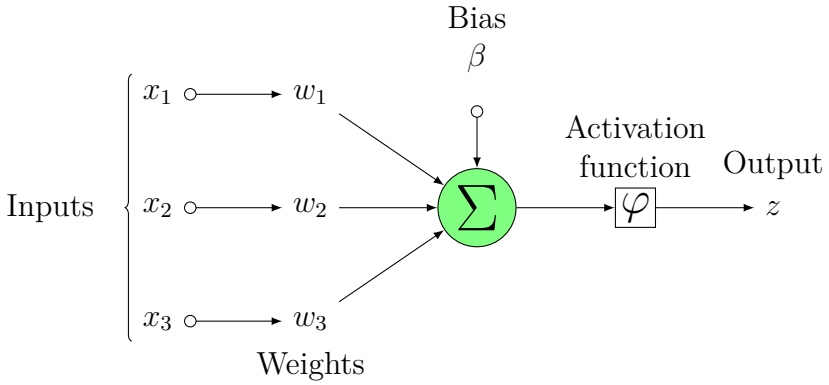

Figure 4.5: This figure illustrates the calculation of each hidden node during the forward propagation. The hidden node is first calculated as a weighted sum of inputs, a bias and weights. Thereafter the node is activated before it is sent towards the next hidden layer.

After the hidden nodes have been calculated and activated, we say that the network has performed a "forward pass". The hidden nodes are activated, as seen in figure 4.5, after the summing of weight and inputs. By activation, we mean something akin to the threshold step-function of the Pitts neuron. Most activation funtions differ from Pitts' binary nature in a sense that they are continuous, and not step-functions. Since there are multiple activation functions to choose from, e.g. tanh, sigmoid, ReLU (see Section 4.6), we will use the symbol φ as our arbitrary activation function symbol.

A forward pass is in short: The creation of the next hidden,or output, layer by multiplying the preceding layer's nodes with the set of weights which connect the two layers. The nodes, in the new hidden layer are then each processed by an activation function

$$z_j = \varphi(h_j). \tag{4.3}$$

The output node, o, is returned from the last forward pass. To attain it, we make a forward pass with the activated hidden nodes of the previous layer, z, along with a new set of unique weights, $w^{(2)}$, connecting the hidden layer and the output node. We can write the expression for the output node in figure 4.4 as

$$o = \sum_{j=1}(z_j w_j^{(2)}) + \beta^{(2)}, \tag{4.4}$$

or

$$o = \sum_{j=1}\left[\sum_{i=1}(x_i w_{ij}^{(1)}) + \beta_j^{(1)}\right] w_j^{(2)} + \beta^{(2)}, \tag{4.5}$$

where o denotes the output node and z_j is the jth activated hidden node. Furthermore, $w_j^{(2)}$ is the weight connecting the jth hidden node to the output node, while $\beta^{(2)}$ is the bias node connected to the output node. Notice that we have changed the name of the weight parameters from w_{ij} to $w_{ij}^{(1)}$ to symbolize which interlayer the weight is located. The bias of the first interlayer has been redubbed similarly from β_j to $\beta_j^{(1)}$.

For fun, let us write the expression for an output node of a network with one more (two) hidden layer

$$o = \sum_{k=1}\left[\sum_{j=1}\left(\sum_{i=1}(x_i w_{ij}^{(1)}) + \beta_j^{(1)}\right) w_{jk}^{(2)} + \beta_k^{(2)}\right] w_k^{(3)} + \beta^{(3)}. \tag{4.6}$$

The expression in eq. (4.6) is not too messy. Yet, trying to differentiate the expressions above, which we later must, is potentially time consuming. We need to switch notation before going any further.

4.4.1 Layer Notation

We want a notation that can generalize to multiple hidden layers. Our weights and biases already supports such notation, e.g. $w_{ij}^{(1)}$ and $\beta_j^{(1)}$. Labeling our nodes' exponents by the layer they belong to, $x \rightarrow x^{(l)}$, provides our example network with these new parameter names: $x_i^{(0)}$, $h_j^{(1)}$, $z_j^{(1)}$. This notation will prove useful for formulating forward propagation in deep neural networks.

4.4.2 Matrix Notation

Matrix notation is a relief when dealing with neural networks, since all layers and weight summation can be represented as vector and matrices products. From eq. (4.2) we have described forward propagation as a sum formula:
$z_j = \varphi\left(\sum_i (x_i w_{ij}^{(1)}) + \beta_j^{(1)}\right)$. To introduce matrix notation, we pack the input nodes and the first set of weights into matrices. For our example network there are two input nodes connected to three hidden nodes, resulting in a 3×2 weight matrix

$$\boldsymbol{w}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} \end{bmatrix}, \mathbf{X} = \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix}, \boldsymbol{\beta}^{(\mathbf{1})} = \begin{bmatrix} \beta_1^{(1)} \\ \beta_2^{(1)} \\ \beta_3^{(1)} \end{bmatrix}. \tag{4.7}$$

(4.8)

With matrix notation it is possible to show that the hidden nodes in eq. (4.2) can be computed with a simple matrix product

$$\begin{aligned} \boldsymbol{w}^{(\mathbf{1})}\mathbf{X} + \boldsymbol{\beta}^{(\mathbf{1})} &= \begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} \\ w_{13}^{(1)} & w_{23}^{(1)} \end{bmatrix} \begin{bmatrix} x_1^{(0)} \\ x_2^{(0)} \end{bmatrix} + \begin{bmatrix} \beta_1^{(1)} \\ \beta_2^{(1)} \\ \beta_3^{(1)} \end{bmatrix} \\ &= \begin{bmatrix} x_1^{(0)} w_{11}^{(1)} + x_2^{(0)} w_{21}^{(1)} + \beta_1^{(1)} \\ x_2^{(0)} w_{12}^{(1)} + x_2^{(0)} w_{22}^{(1)} + \beta_2^{(1)} \\ x_3^{(0)} w_{13}^{(1)} + x_2^{(0)} w_{23}^{(1)} + \beta_3^{(1)} \end{bmatrix} = \begin{bmatrix} h_1^{(1)} \\ h_2^{(1)} \\ h_3^{(1)} \end{bmatrix} \\ &= \mathbf{h}^{(1)}. \end{aligned} \tag{4.9}$$

The activated values, $z_j^{(1)}$, can now be written as

$$\mathbf{z}^{(1)} = \varphi(\mathbf{h}^{(1)}) = \varphi(\boldsymbol{w}^{(1)}\mathbf{X} + \boldsymbol{\beta}^{(1)}), \tag{4.10}$$

where the activation function operates elementwise on the hidden nodes, $\mathbf{h}^{(1)}$, giving $\mathbf{z}^{(1)}$ the same length as $\mathbf{h}^{(1)}$.

Next up is the output layer/node. Since there is only one output node in our example, we will treat the output value as a scalar. The expression for the output

node has already been described in eq. (4.4) as $o = \sum_j (z_j^{(1)} w_j^{(2)}) + \beta^{(2)}$, but for the sake of generalization (and later derivations), let us introduce an activation function in the output layer: $o = z^{(2)} = \varphi^*(\sum_j (z_j^{(1)} w_j^{(2)}) + \beta^{(2)})$. In order to not mess with our model, φ^* is simply the linear activation function: $\varphi^*(x) = x$. Since each activated node is named z, the output node of our example network is now dubbed $z^{(2)}$:

$$\begin{aligned}
\mathbf{z}^{(2)} &= \varphi^* \left(\boldsymbol{w}^{(2)} \mathbf{z}^{(1)} + \beta^{(2)} \right) \\
&= \varphi^* \left(\begin{bmatrix} w_1^{(2)} & w_2^{(2)} & w_3^{(2)} \end{bmatrix} \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \end{bmatrix} + \begin{bmatrix} \beta^{(2)} \end{bmatrix} \right) \\
&= \varphi^* \left(\begin{bmatrix} w_1^{(2)} z_1^{(1)} + w_2^{(2)} z_2^{(1)} + w_3^{(2)} z_3^{(1)} + \beta^{(2)} \end{bmatrix} \right) \\
&= \varphi^* \left(\begin{bmatrix} h_1^{(2)} \end{bmatrix} \right) \\
&= \begin{bmatrix} z_1^{(2)} \end{bmatrix}.
\end{aligned} \tag{4.11}$$

The unactivated output node, which we here unintuitively call $h^{(2)}$, is activated by the linear activation function φ^*. Doing this gives us a consistent way to express backward propagation later on in this thesis.

The last thing we want to do in this section is to use eq. (4.11) to make a generalized way of computing the lth layer of nodes in any feed forward neural network from the l-1th layer

$$\begin{aligned}
&\mathbf{z}^{(2)} = \varphi^* \left(\boldsymbol{w}^{(2)} \mathbf{z}^{(1)} + \beta^{(2)} \right) \\
&\Longrightarrow \\
&\mathbf{z}^{(l)} = \varphi \left(\boldsymbol{w}^{(l)} \mathbf{z}^{(l-1)} + \beta^{(l)} \right).
\end{aligned} \tag{4.12}$$

We now have a matrix multiplication scheme that makes for a much more compact notation and algorithm. Keep in mind that for a neural network of n layers (including the input layer and the output layer), the last/n-1th forward propagation should have φ^* as its activation function, since $z^{(n-1)}$ is the output node. Additionally, $l \in \{2, ..., n-1\}$ because $z^{(0)}$ does not exist. In order to find the first layer of activated nodes, $\mathbf{z}^{(1)}$, we must use eq. (4.10)

$$\mathbf{z}^{(1)} = \varphi(\boldsymbol{w}^{(1)} \mathbf{X} + \boldsymbol{\beta}^{(1)}).$$

4.5 Initialization

Having already talked about how data flows through a neural net, we have yet to discuss how one is created. This section introduces standard ways of how one would initialize a neural network's weights and bias nodes. A newly initialized neural

network is next to useless without having been trained, but the initialization method can have large consequences for the training duration. In addition to discussing the effects of certain initialization methods, this section will also briefly touch on the effects of a neural network's size.

4.5.1 Layers and Nodes

Before deciding how many layers and nodes you would like for your network, it's important to know their effect on the network. Vaguely put, additional layers and nodes serve to increase the flexibility of a neural network. By flexibility we mean the neural network's ability to fit a regression line through our data. Just like flexible regression methods, neural networks also risk overfitting when given enough nodes and layers. Since there are multiple methods to bypass overfitting when training a neural network, one is almost guaranteed to not fall into the trap of overfitting their model.

The computational complexity of training and using a neural network is directly dependent on the number and nodes and layers in your network. Assuming you have an input layer of n nodes, n layers, excluding the output, and n nodes in each hidden layer and n output nodes, the complexity of the forward propagation becomes $\mathrm{O}(n^4)$. To derive this, we start with the input layer of length n, which is multiplied with a square weight matrix of dimensions $n \times n$. This forward propagation outputs n hidden nodes, which then receive n activation operations, one for each hidden node. This forward pass happens n times since we have n layers.

$$\begin{aligned} n_{forward\_pass} &\approx n_{layers} \times (n_{matrix\_mul} + n_{\varphi}) \\ &= O(n)(O(n^3) + O(n)) \\ &= O(n^4) + O(n^2) \end{aligned} \tag{4.13}$$

In eq. (4.13), n_{φ} is the number of elements the activation function has to operate on, while n_{matrix_mul} is the number of multiplication operations when multiplying a vector with a square matrix. The approximation sign is there due to us ignoring the bias nodes, which make a negligible contribution to the complexity.

4.5.2 Weight Initialization

The number of weights are defined by the number of layers and nodes you have decided to go with. The number of weights grow qudratically with n, when n is the number of layers and nodes making it normal to find hundreds of thousands of weights in large neural networks. These weights are the actual parameters of a neural network, as they are doing all the learning, while the nodes simply are their sums.

The initialization of weights can be divided into two categories, uniform and normally distributed weights. There are many methods withing these two categories, but they all have in common that they draw numbers from random distributions and use these to set the weights' values.

- **All Zero Initialization (etc.)** is the initialization of all weights to zero, which belongs to neither of the aforementioned categories. It is not advised to initialize the weights to zero. Although it is reasonable to assume that in a fully trained network, half its weights will be negative and half of them will be positive, setting our weights to zero can be thought of as a good approximation. The problem with this initialization is that all weights will be treated similarly during the backpropagation algorithm (see Section 4.7.3), meaning they will all be updated in an identical manner. This causes a complete lack of asymmetry between neurons, disallowing any form of learning. The reason why is because all our parameters are acting as one, removing all flexibility from the model.

- **Uniform Distribution Initialization** Uniform probability distributions are utilized in multiple heuristics for initializing a neural net's weights. In some of these methods, weights are drawn from a distribution, $\boldsymbol{U}\left[-a, a\right]$, where a is some function of the number of nodes in the previous layer

  $$a(d_{in}) = c/\sqrt{d_{in}}. \tag{4.14}$$

  Some ways to go about designing these distributions, like Fahlman (1988) did, is to manually find the best uniform distribution through testing. By best, we mean the distribution which consistently returns weights that end up helping the neural network learn the fastest.

- **Xavier initialization** was developed with deeper networks in mind. Deep neural networks struggle with what we call the "vanishing gradient problem", as well as the "exploding gradient problem". In Glorot and Bengio (2010), the authors illustrate how certain weight initialization methods, in combination with sigmoidal activation functions, block the flow of information by making the nodes converge towards zero. This is the "vanishing gradient problem". The authors' illustration, figure 4.6, illustrates both the vanishing gradient problem and how their initialization method solves the problem.

  To begin initializing the weights, we begin with a normal distribution $\mathcal{N}(0, 1)$ - centered around zero with a standard deviation of 1. Second, we fill all our weight matrices with number from this distribution. We then calculate the variance of each weight matrix:

  $$var(\boldsymbol{w}^{(l)}) = \frac{2}{d_{in} + d_{out}}$$

  where $\boldsymbol{w}^{(l)}$ is the lth weight matrix, d_{in} is the length of the previous layer, and d_{out} is the length of the next one. The next step is to create a customized

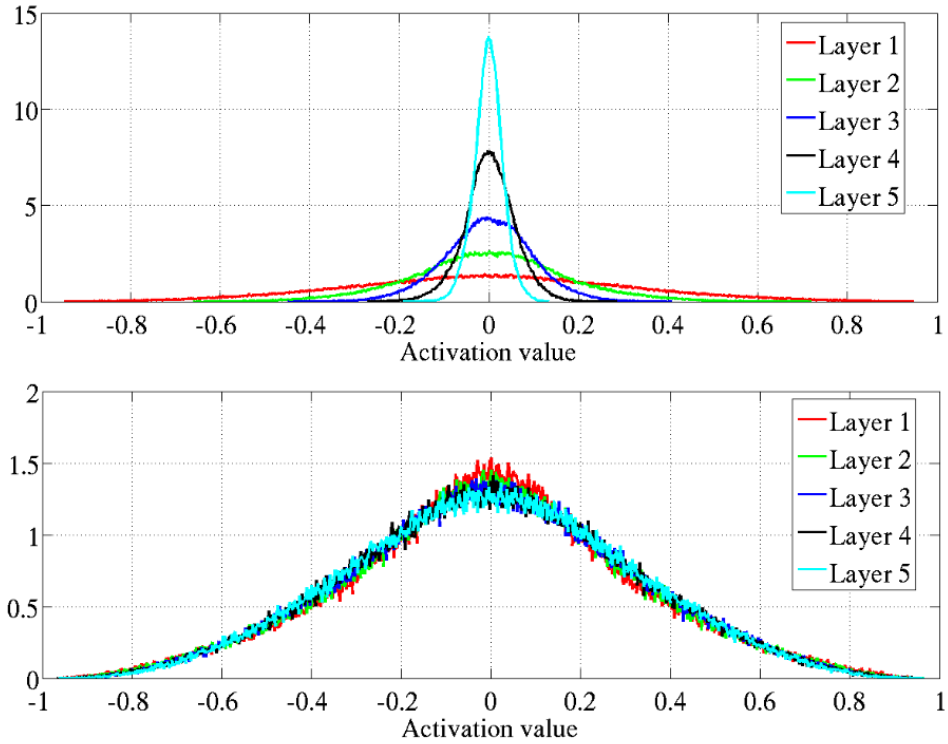

Figure 4.6: Both plots contain five scaled histograms. The histograms show how the values of activated hidden nodes from five individual hidden layers of a neural network are distributed. Top: Trained neural network which uses tanh activation and uniform distribution initialization. Bottom: Same network, except the weights have been Xavier initialized. Notice how the signal's distribution stays nearly identical along all five forward passes, meaning there are less vanishing signals. Figures taken from Glorot and Bengio (2010)

normal distribution, $\mathcal{N}(0, 1/\sqrt{var(\boldsymbol{w}^{(l)})})$ for each weight matrix and fill each with values drawn from their custom distribution. These distributions give the weight matrices a variance that scales with the number of weights inside of it. As a consequence, the forward and backward propagated matrix products (hidden layers) get an estimated variance of 1. This results in propagated signals neither diminishing nor exploding.

4.5.3 Bias Initialization

There is not much to be said about bias initialization. Typically, one sets all bias weights to zero. This does not have any negative consequences as long as the weights are initialized asymmetrically. A variance in the weight values allows the weight update algorithm (See backpropagation in section 4.7.3) to work properly. This algorithm is able to adjust both the weights and the bias so that our error function can reach a better minima. Although zero initialization is most popular, one can treat bias initialization as a hyperparameter and attempt to optimize bias initialization to get better learning results by e.g. using random numbers from a uniform or normal distribution instead of zeros.

4.6 Activation Functions

Without activation functions, an artificial neural network is no more than a combination of linear products between the inputs and weights of the network. In order to mimic an arbitrary non-linear function, neural networks need to possess a non-linear activation function.

Activation functions like the step function of the Pitts neuron are made to model the firing rate of biological neurons' activation potential. Unlike the step function of the Pitts neuron, most activation functions are continuous, with continuous derivatives. A notable exception to activation functions with continuous derivatives is the ReLU function.

4.6.1 ReLU

"ReLU" stands for **rectified linear unit** because it functions as a unit which rectifies its input. A rectifier is a function which returns its positive arguments and nullifies any negative ones

$$\varphi_{ReLU} = x^{+} = \texttt{max}(0, x). \tag{4.15}$$

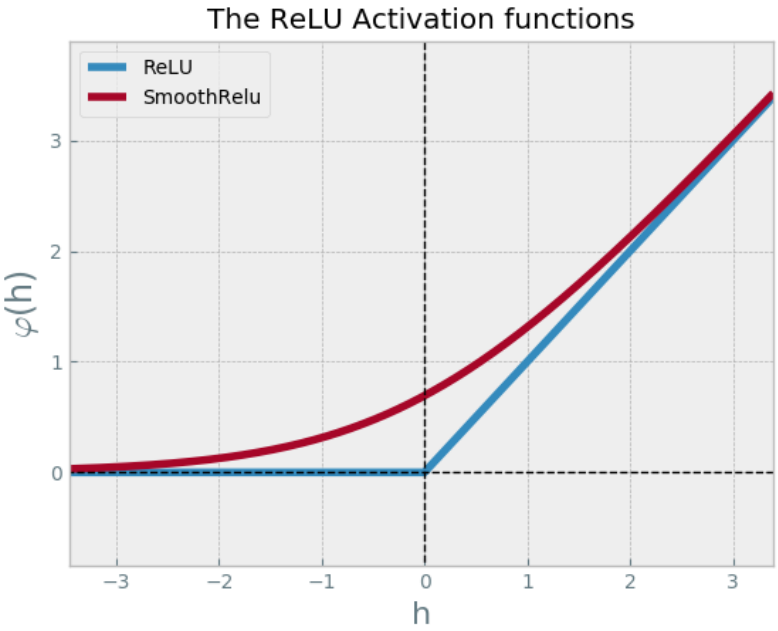

Figure 4.7: ReLU, short for **rectified linear unit**, is an activation function that only returns the positive part of its argument. Other variants of ReLU, such as SmoothReLU exist. The advantage of the latter is its continuous derivative. When we reach the section on backpropagation, we will explain why a continuous derivative can be a plus.

In figure 4.7, you can observe how ReLU behaves when activating hidden node values. Contrary to sigmoidal activation functions, ReLU does not diminish positive signals. The consequence of using ReLU as an activation function is the same as for Xavier initialization, namely that vanishing gradients have a lower chance of taking place in the later layers. This leads to quicker learning in deep neural networks (Krizhevsky, Sutskever, and E. Hinton 2012).

The ReLU activation function and its derivative are much quicker to calculate than their sigmoidal cousins. Since ReLU is zero all the way until $x > 0$, and linear afterwards, we can define its derivative function as :

$$\varphi'_{ReLU}(x) = \begin{cases} 0 & x < 0 \\ \texttt{undefined} & x = 0 \\ 1 & \text{if } x > 0. \end{cases}$$

4.6.2 Sigmoids

Using an activation function of the sigmoid family in your neural network causes the hidden node values to be mapped to a range, usually between -1 and 1 or 0 and 1. A more formal definition of the sigmoids is; a differentiable real function,

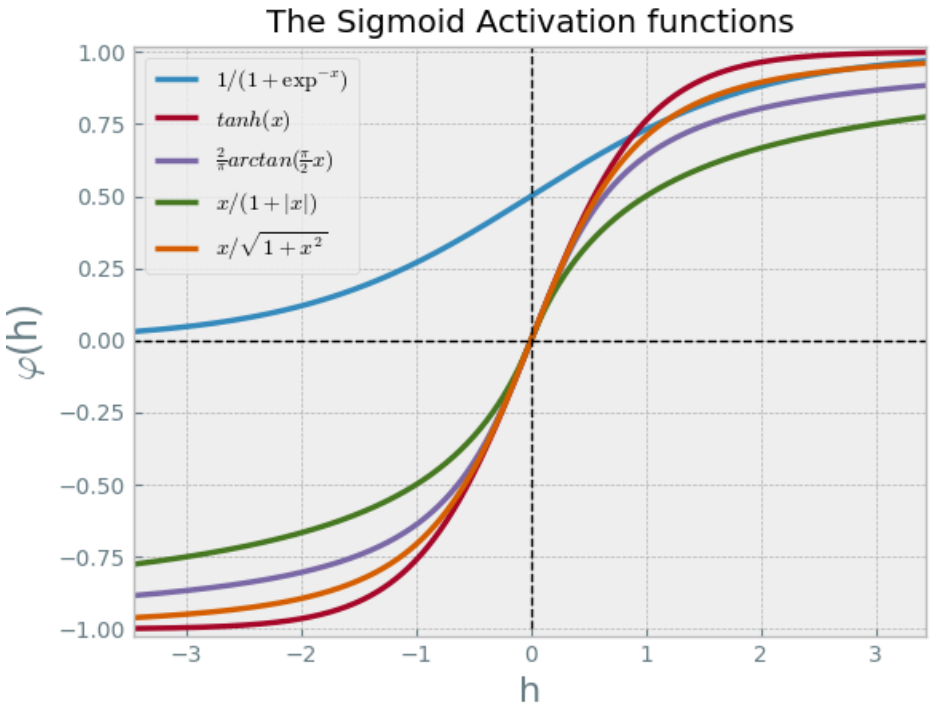

Figure 4.8: Above is a depiction of various sigmoids. The two first labels are likely the most popular among sigmoid activation functions. Both this image and figure 4.7 are based on graphs made by Georg-Johann. This figure specifically is based on a figure from Johann (2010).

defined for all real inputs, with a positive derivative at each point (Han and Moraga 1995). Infinitely many such functions and some of the most popular variants used in machine learning is seen in figure 4.8.

Sigmoids are such that if they receive increasingly large inputs, the output values saturate by converging towards 1. This effect also happens for increasingly large negative input, which result in outputs converging toward 0 or -1. This saturation is positive and negative for learning. On one hand, sigmoids make it harder for the neural networks' nodes to diverge, the explosive gradient problem, since all the nodes are attentuated by the sigmoid activation. On the other hand, this squashing effect can induce the vanishing gradient problem. The asymptotic saturation also removes the neural networks ability to distinguish between large inputs and even larger inputs. This reduces a network's continued ability to learn.

The sigmoid function we have used in this thesis is the sigmoid

$$\sigma(x) = \frac{1}{1+\exp(-x)}. \tag{4.16}$$

Going forward in this thesis, the term "sigmoid" will always refer to the function in

eq. (4.16). Since we need the derivative of the sigmoid in order to backpropagate later, I will do a quick derivation of its derivative here

$$\frac{\partial \sigma(x)}{\partial x} = \frac{\partial}{\partial x}\left[\frac{1}{1+\exp(-x)}\right] \tag{4.17}$$

$$= \frac{\partial}{\partial x}\left(1+\exp(-x)\right)^{-1} \tag{4.18}$$

$$= -(1+\exp(-x))^{-2}(-\exp(-x)) \tag{4.19}$$

$$= \frac{1}{(1+\exp(-x))}\frac{\exp(-x)}{(1+\exp(-x))} \tag{4.20}$$

$$= \frac{1}{1+\exp(-x)}\frac{(1+\exp(-x))-1}{1+\exp(-x)} \tag{4.21}$$

$$= \frac{1}{1+\exp(-x)}\left[\frac{1+\exp(-x)}{1+\exp(-x)} - \frac{1}{1+\exp(-x)}\right]. \tag{4.22}$$

Substituting eq. (4.16) into eq. (4.22) gives us

$$= \sigma(x)(1-\sigma(x)). \tag{4.23}$$

As we can see, in the case of the sigmoid function, its derivative can be rewritten in terms of itself.

4.6.3 Bias and Activation

Adding bias to the hidden nodes after they have been calculated has an effect on the threshold of the subsequent activation. Activation functions are similar to threshold functions, especially ReLU(see figure 4.7). For ReLU, input signals which are negative are blocked, while positive signals are allowed through. What bias offers is a way to control the activation function's threshold so that input signals that previously would get squashed to zero are allowed to pass through, or oppositely. The figures in 4.10 detail how bias can move the threshold of activation, changing the nature of signal flow throughout the neural network.

4.7 Training the Neural Network

Although we have not yet talked about what we mean by learning, we have defined all the variables which require learning, i.e. weights and biases. Learning, training

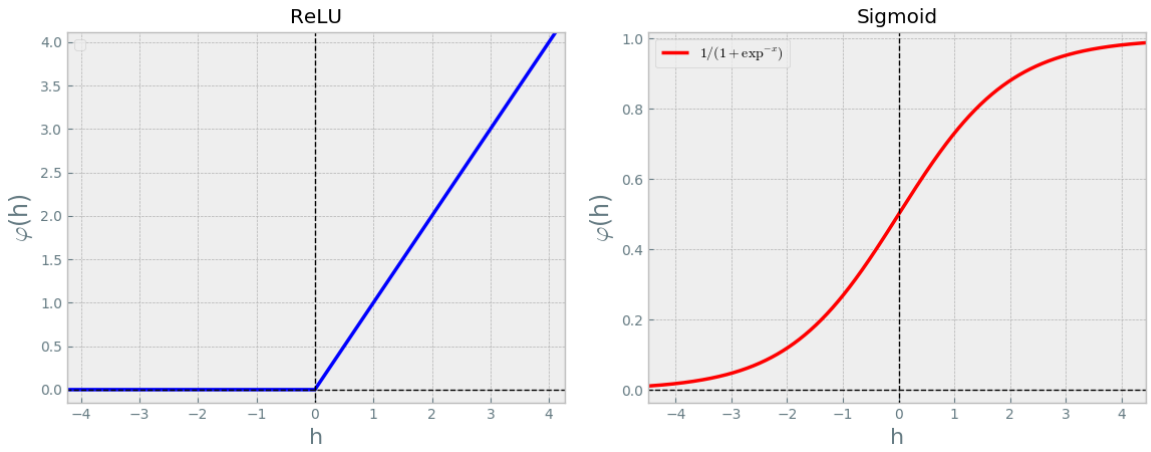

Figure 4.9: When a neural network has no trainable bias, the activation functions behave as the static threshold-like functions seen above.

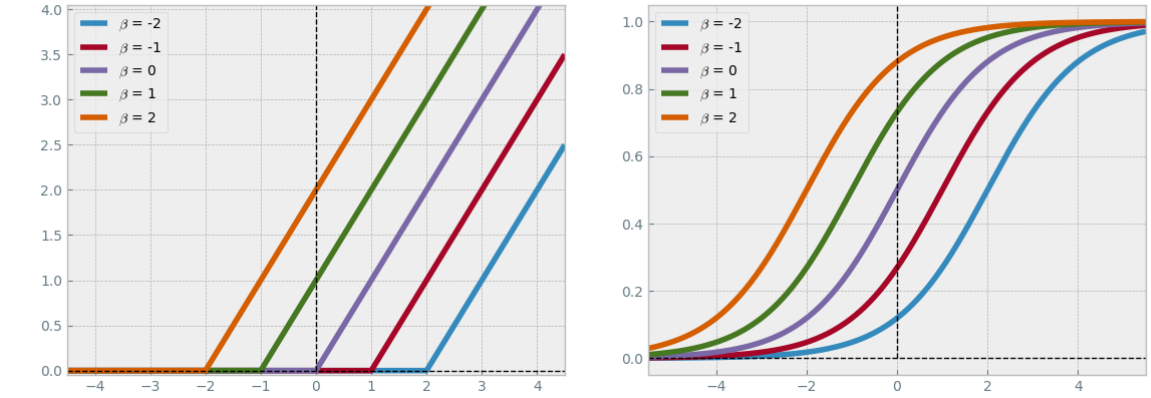

Figure 4.10: As the bias node changes, we see that it affects the activation of the hidden nodes. Negative bias push the threshold, or center, of the activation functions forward, raising the bar for what it means to be a strong signal. While positive bias increase the weaker signals' chance to successfully propagate.

and fitting are all synonymous with each other and refer to the fitting of a model's parameters. Training a neural network model is similar to fitting a regression model, which we touched upon in section 3.3. Compared to the two parameters of a simple linear regression model, neural networks with their thousands of parameters are much harder to optimize. In this section we will repeat some basics of training regression models, introduce the loss function, followed by an introduction of the backpropagation algorithm.

4.7.1 Data Split

In section 3.3 we introduced the splitting of data into training- and test datasets before training regression models. With neural networks on the other hand, it is common to split the data ($\mathbf{X}$,$\mathbf{y}$) into three non-overlapping subsets: a training dataset ($\mathbf{X}_{train}$,$\mathbf{y}_{train}$), a test dataset ($\mathbf{X}_{test}$,$\mathbf{y}_{test}$) and a validation dataset ($\mathbf{X}_{val}$,$\mathbf{y}_{val}$). In most regression problems, $\mathbf{y}$, is a simple vector of N label values. $\mathbf{X}$ on the other hand, is an $N \times M$ matrix. $\mathbf{X}_i$, where $i \in \{1, 2, ..., N\}$, are input vectors of length M. The point of a regression model is to output that best matches the target value $\mathbf{y}_i$ by simply feeding the model batches of $\mathbf{X}_i$ while

As training progresses, the neural network becomes better and better at modeling data it has already seen. But, the point of the model is to infer it on data which it has not yet seen, which is why we made the test dataset. Using the test set, we can evaluate our model's ability to map $\mathbf{X}_{test}$ to $\mathbf{y}_{test}$. However, there will likely be a discrepancy between how well it performs on the training data and the test data. This is due to the potential of overfitting (see section 3.4). In order to circumvent overfitting, we stop training when the model's ability to fit the validation dataset starts to decrease. We will come back to this in section 8.3, which is after we have discussed exactly what it means to train a neural network.

4.7.2 Cost Functions

An important part of training a neural network is its cost function. A cost function, also called loss function, is the function that helps define our learning problem as a minimization problem. The function shows how far off a neural network's prediction is. The "targets", or $\mathbf{y}$, is what we want our neural network to predict.

The prediction of our example neural network is the same as its output node, $z^{(2)}$. For the sake of generalization, let us call the ouput node $z^{(o)}$. How far off a model's prediction is, is represented by a residuals. We introduced residuals in the section on OLS 3.2.2. Similarly to neural networks, OLS minimizes a cost function, sum of squared residuals, in order to train a model.

The residual of a prediction w.r.t its target is the difference between them

$$r_i = y_i - z_i^{(o)}(\mathbf{X}_i).$$

Here, $z_i^{(o)}(\boldsymbol{X}_i)$ is our model's output for the ith training example ($\mathbf{X}_i,\ y_i$) where $\mathbf{X}_i$ serves as the input layer and y_i as the target.

After calculating the error residuals r_i over multiple training examples, we usually accumulate them. How one does this depends on which cost function the neural network uses. Below we present **MSE** along with two other popular loss functions.

- **MSE**, the **mean squared error**, is an aggregation function whose output is the mean of the squared residuals. In this thesis, we have exclusively used the MSE as loss function and so, MSE is implied going forward.

$$MSE(w, \beta) = \frac{1}{2N} \sum_{i}^{N} (y_i - \hat{y}_i(\mathbf{X}_i))^2$$

  The reason why we add a "2" in the denominator is so that the expression of the derivative of the cost function becomes prettier. Just like in OLS regression, the backpropagation algorithm minimizes the cost function by first finding and exploiting its derivatives.

- **MAE**, or the **mean absolute error**, is a loss function very similar to **MSE**, the only difference being that we don't square the residuals (i.e. l1 norm). This means outliers are not weighted as heavily.

$$MAE(w, \beta) = \frac{1}{N} \sum_{i}^{N} |(y_i - \hat{y}_i)|$$

- **Cross-Entropy** loss is a popular loss function used in categorization problems, which is not what we are doing in this thesis. For categorization problems, there are usually multiple output nodes in a network. The jth output node represents the network's certainty of the ith training example belonging to the jth class, e.g. cat, dog or bird. The output nodes, or predictions, are intuitively called p_{ij}, while the label y_{ij} is either 1 or 0, e.g. cat or not cat. Having set up our notation, we can use Cross-Entropy which measures the dissimilarity between the prediction p_{ij} and the truth y_{ij}:

$$CE(w, \beta) = \frac{1}{N} \sum_{i}^{N} \left[- \sum_{j}^{K} y_{ij} \log(p_{ij}) \right]$$

4.7.3 Backpropagation

The basics of the backpropagation algorithm was first derived by Kelley (1960) and Bryson (1961). The full potential of this algorithm was not realized until some 30 years after its invention, when its popularity started to grow for real. This is likely due to computational resources being too limited for the algorithm to run on a network of sufficient size at the time. The goal of this section is to create an intuitive understanding of backpropagation, an algorithm that computes partial derivatives, or "gradients" of a loss function, by using the chain rule recursively. Onwards, the word gradient will refer to the derivative of a node in a neural network, e.g. $z_j^{(l)}$, w.r.t. the loss, L.

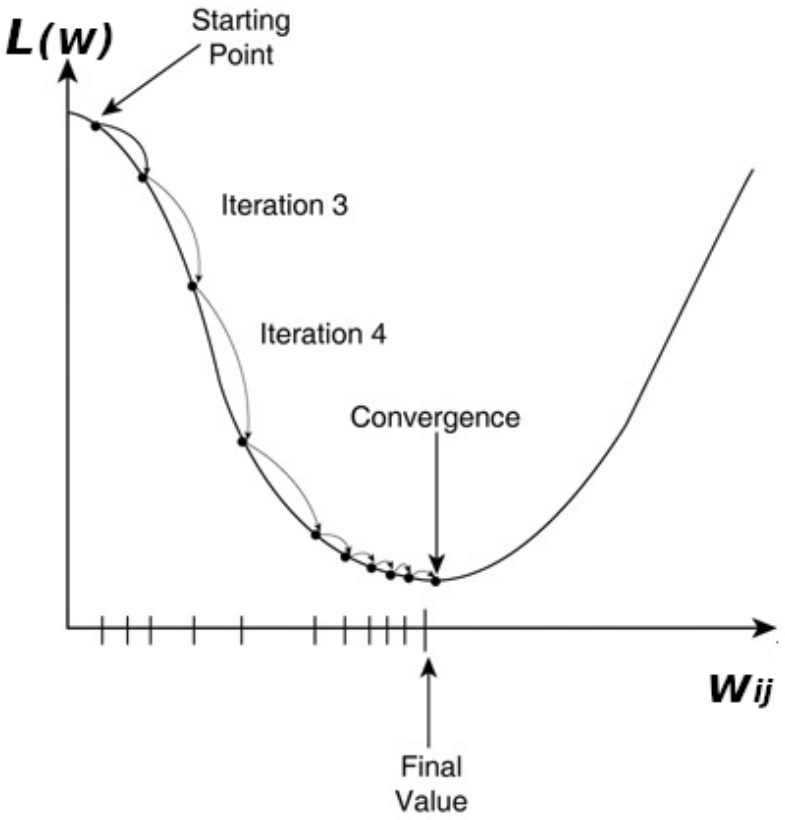

Figure 4.11: The figure shows the minimization of our loss function L by tweaking the weight w_{ij}. Using gradient descent, we know which way to tweak w_{ij} since we know the sign of the loss function's derivative. The figure is taken from Jiaconda (2016).

Gradient Descent

Backpropagation is a form of gradient descent. Gradient descent uses the gradient of a loss function with respect to one of its parameters to minimize the loss function. This minimization is possible because the gradient of the loss function reveals whether to increase or decrease said parameter in order to decrease the loss. For a visual representation of gradient descent, see figure 4.11. Similarly to how we solved the linear regression problem with OLS, we can minimizing the **MSE** loss function with respect to the weights and biases of the network

$$L(\beta^{(1)}, \beta^{(2)}, w^{(1)}, w^{(2)}) = \frac{1}{2N} \sum_{i}^{N} (y_i - z^{(2)}(\mathbf{X}_i))^2, \tag{4.24}$$

$$\min_{\boldsymbol{w}^{(1)}, \boldsymbol{w}^{(2)}, \boldsymbol{\beta}^{(1)}, \boldsymbol{\beta}^{(2)}} L(\mathbf{X}_i). \tag{4.25}$$

As the name suggests, gradient descent requires gradients, or partial derivatives,

of the loss in order to minimize the loss:

$$\frac{\partial L(\mathbf{X})}{\partial w_{ij}^{(1)}}, \frac{\partial L(\mathbf{X})}{\partial w_{j}^{(2)}}, \frac{\partial L(\mathbf{X})}{\partial \beta_{j}^{(1)}}, \frac{\partial L(\mathbf{X})}{\partial \beta^{(2)}}$$

When a derivative is computed, gradient descent changes the parameter (e.g. a weight $w_j^{(2)}$) in the negative direction of its derivative

$$w_j^{(2)} := w_j^{(2)} - \alpha \frac{\partial L(\mathbf{X})}{\partial w_j^{(2)}}. \tag{4.26}$$

This is the same as "updating" the network, or "training". The coefficient of the gradient, α, is called the "learning rate". There exists multiple techniques for how to dynamically change the learning rate in order to reach a best minimum for the loss as possible (see section 4.7.4). In figure 4.11, the learning rate can be thought of as the distance the loss changes between each step of the algorithm.

Numerical Gradients

Backpropagation gradient descent uses the analytical gradient instead of the numerical gradient of the loss. Numerical gradients are approximations and are found using methods such as two point estimation:

$$\frac{\partial L(w_j^{(2)})}{\partial w_j^{(2)}} \approx \lim_{\epsilon \to 0} \frac{L(w_j^{(2)}) - L(w_j^{(2)} + \epsilon)}{2}$$

Where ϵ is a small increment in $w_j^{(2)}$. This method requires two forward propagations for every parameter, because we need to obtain both $L(w_j^{(2)})$ and $L(w_j^{(2)} + \epsilon)$ in order to update $w_j^{(2)}$. For a network of n layers with n nodes per layer, we found that forward propagation has a computational complexity of $O(n^4)$ (see section 4.5.1). Such a network has n^2 weights between each layers, making up n^3 weights in total. To update n^3 weights with numerical gradients, we would have to forward propagate the network at least twice for each weight, resulting in a very heavy computation: $O(n^3 \times 2 \times n^4) \approx O(n^7)$. Let us rather derive how to find the gradients analytically.

Analytical Gradients

Analytical gradients are faster to compute and more accurate compared to numerical gradients. The only downside to analytical gradients is that if one is to implement a neural network from the ground up, one has to know how to differentiate a complex function like a neural network. Luckily for us, we have used frameworks like

Tensorflow/Keras which do these tedious things for us. In this section we present how to derive the analytical gradient of a parameter in our simple example network. We then generalize these steps to create an algorithm that can update any of the weights in our network.

To derive an expression for the gradient of a weight, $w_j^{(2)}$, we backpropagate its derivative w.r.t. the error, L, by expanding the derivative with the chain rule

$$\frac{\partial L}{\partial w_j^{(2)}} = \frac{\partial L}{\partial h^{(2)}} \frac{\partial h^{(2)}}{\partial w_j^{(2)}}. \tag{4.27}$$

We can easily find each of the partial derivatives on the right hand side of eq. (4.27) independently

$$\frac{\partial L}{\partial h^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(2)}} \tag{4.28}$$

$$\begin{aligned} \frac{\partial L}{\partial z^{(2)}} &= \frac{\partial}{\partial z^{(2)}} \frac{1}{2}(y - z^{(2)})^2 \\ &= -(y - z^{(2)}) \frac{\partial z^{(2)}}{\partial z^{(2)}} \\ &= z^{(2)} - y \end{aligned} \tag{4.29}$$

$$\begin{aligned} \frac{\partial z^{(2)}}{\partial h^{(2)}} &= \frac{\partial \varphi^*(h^{(2)})}{\partial h^{(2)}} \\ &= 1 \implies \end{aligned} \tag{4.30}$$

$$\begin{aligned} \frac{\partial L}{\partial h^{(2)}} &= z^{(2)} - y \\ &= \delta^{(2)}. \end{aligned} \tag{4.31}$$

The gradient of the error, L, with respect to the node $h_k^{(l)}$ will from here be written as $\delta_k^{(l)}$ where l and k denotes which layer and where in the layer the node resides. The output node $h^{(2)}$ is alone and so the error gradient of the output node $\delta^{(2)}$ requires no index k. To find the gradients of parameters further back in the network, we must propagate this gradient backward. But first, let us finish updating the weight $w_j^{(2)}$ by finding the last derivative in eq. (4.27)

$$\frac{\partial h^{(2)}}{\partial w_j^{(2)}} = \frac{\partial}{\partial w_j^{(2)}} \left(\sum_{j'} w_{j'}^{(2)} z_{j'}^{(1)} + \beta^{(2)} \right). \tag{4.32}$$

As the sum function in eq. (4.32) iterates over the nodes in the last layer, eventually j' will match the index j. For all iterations where j' and j do not match, the derivative expression returns zero

$$\frac{\partial}{\partial w_j^{(2)}} \left(\sum_{j' \neq j} w_{j'}^{(2)} z_{j'}^{(1)} + \beta^{(2)} \right) = 0, \tag{4.33}$$

but when they finally match we get a non-zero result

$$\begin{aligned}\frac{\partial h^{(2)}}{\partial w_j^{(2)}} &= \frac{\partial}{\partial w_j^{(2)}}\left(w_{j'=j}^{(2)} z_{j'=j}^{(1)} + \beta^{(2)}\right) \\ &= \frac{\partial}{\partial w_j^{(2)}} w_j^{(2)} z_j^{(1)} + \beta^{(2)} \\ &= z_j^{(1)}. \end{aligned} \tag{4.34}$$

The gradient of $w_j^{(2)}$ can now be expressed as the gradient of its subsequent node and the preceding activated node

$$\frac{\partial L}{\partial w_j^{(2)}} = \delta^{(2)} z_j^{(1)}. \tag{4.35}$$

After a successful forward and backward propagation, and having finally attained the gradient in eq. (4.35), we can now update the parameter $w_j^{(2)}$ by subtracting the gradient from the parameter, as seen in eq. (4.26):

$$w_j^{(2)} := w_j^{(2)} - \delta^{(2)} z_j^{(1)}.$$

The gradient of a bias node $\beta^{(2)}$ is quite similar to derive as a gradient of a weight from the same interlayer ($w_j^{(2)}$). The only difference from the gradient expressions is their last factor

$$\begin{aligned}\frac{\partial L}{\partial w_j^{(2)}} &= \delta^{(2)} \frac{\partial h^{(2)}}{\partial w_j^{(2)}} \\ \frac{\partial L}{\partial \beta^{(2)}} &= \delta^{(2)} \frac{\partial h^{(2)}}{\partial \beta^{(2)}}. \end{aligned} \tag{4.36}$$

The last factor of the bias gradient always equals one due to the differentiation canceling all other terms out

$$\begin{aligned}\frac{\partial h^{(2)}}{\partial \beta^{(2)}} &= \frac{\partial}{\partial \beta^{(2)}}\left[\sum_{j'}\left(w_{j'}^{(2)} z_{j'}^{(1)}\right) + \beta^{(2)}\right] \\ &= 1 \implies \\ &= \frac{\partial L}{\partial \beta^{(2)}} = \delta^{(2)}. \end{aligned} \tag{4.37}$$

In eq. (4.37) we can see the error gradient of the bias connected to a node $h_k^{(l)}$ is always equal to the gradient of said node, $\delta_k^{(l)}$. From eq. (4.35) we can conclude that the gradient of a weight $w_{jk}^{(l)}$ is equal to the gradient of the node $h_k^{(l)}$ multiplied

with the activated node in the preceding layer, $z_j^{(l-1)}$

$$\frac{\partial L}{\partial w_{jk}^{(l)}} = \delta_k^{(l)} z_j^{(l-1)}, \tag{4.38}$$

$$\frac{\partial L}{\partial \beta_k^{(l)}} = \delta_k^{(l)}. \tag{4.39}$$

Here, j is the index of nodes in preceding layer, while k is the index of nodes in the subsequent layer.

Equations 4.38 and 4.39 both depend on the error gradient of the nodes, $\delta_k^{(l)}$. In order to update a weight in the second interlayer, we first had to find $\delta^{(2)}$, and likewise, to update a weight in the first interlayer, we are going to need $\delta_j^{(1)}$. By $\delta_j^{(1)}$, we mean the error gradient of $h_j^{(1)}$. To find the error gradient of a node that precedes our obtained error, $\delta_{(2)}$, we must backpropagate the error, L, another step

$$\begin{aligned}
\delta_j^{(1)} &= \frac{\partial L}{\partial h_j^{(1)}} \\
&= \sum_k^1 \frac{\partial L}{\partial h_k^{(2)}} \frac{\partial h_k^{(2)}}{\partial h_j^{(1)}} \\
&= \sum_k^1 \delta_k^{(2)} \frac{\partial h_k^{(2)}}{\partial z_j^{(1)}} \frac{\partial z_j^{(1)}}{\partial h_j^{(1)}}.
\end{aligned} \tag{4.40}$$

Let us remind ourselves that there is only one output node in our example network, while the preceding layer of the output has 3 nodes, so $k \in \{1]\}$ and $j \in \{1, 2, 3\}$. We have indexed the output node so that our equations can be generalized to any layer

$$\delta_j^{(l-1)} = \sum_k^1 \delta_k^{(l)} \frac{\partial h_k^{(l)}}{\partial z_j^{(l-1)}} \frac{\partial z_j^{(l-1)}}{\partial h_j^{(l-1)}}. \tag{4.41}$$

From eq. (4.30) we see that the last factor in eq. (4.41), $\partial z_j^{(l-1)}/\partial h_j^{(l-1)}$, can be written as $z'^{(l-1)}(h_j^{(l-1)}) \to z_j'^{(l-1)}$. The last derivative in eq. (4.41), $\partial h_k^{(l)}/\partial z_j^{(l-1)}$, can be rewritten as the weights connecting the subsequent layer with a node in the preceding layer

$$\frac{\partial h_k^{(l)}}{\partial z_j^{(l-1)}} = \frac{\sum_{j'} w_{j'k}^{(l)} z_{j'}^{(l-1)} + \beta_k^{(l)}}{\partial z_j^{(l-1)}} \tag{4.42}$$

$$= w_{jk}^{(l)}. \tag{4.43}$$

With these two new finding we now rewrite the formula for the backpropagation of the node gradients in eq. (4.41) as a much more readable expression

$$\delta_j^{(l-1)} = z_j'^{(l-1)} \sum_k^1 w_{jk}^{(l)} \delta_k^{(l)}, \tag{4.44}$$

where $l \in \{2, ..., n-1\}$, for a neural network with n layers, including input and output layers.

Matrix Notation

Like forward propagation, backward propagation can be written as a series of matrix operations. In section 4.4.2 we showed that

$$\sum_i x_i w_{ij}^{(1)} \rightarrow \boldsymbol{w^{(1)}}\mathbf{X}, \tag{4.45}$$

and by extension, the sum in eq. (4.44), where the sum index of the weight matrix is transposed, can be written as

$$\sum_k w_{jk}^{(l)} \delta_k^{(l)} \rightarrow \boldsymbol{w}^{(l)T}\boldsymbol{\delta}^{(l)}, \tag{4.46}$$

where the weights in the backpropagation are the same weights used in the forward propagation, but transposed. The product in eq. (4.46) returns the gradients of the nodes in the l-1th layer, $\delta_j^{(l-1)}$, except that they also have to be multiplied with $z_j'^{(l-1)}$. To multiply the factors $z_j'^{(l-1)}$ with our nodes $w_{jk}^{(l)}\delta^{(l)}$, as is done in eq. (4.44), we can use the Hadamard product

$$\begin{aligned} \delta_j^{(l-1)} &= z_j'^{(l-1)} \sum_k^1 w_{jk}^{(l)} \delta_k^{(l)} \rightarrow \\ \boldsymbol{\delta}^{(l-1)} &= \mathbf{z}'^{(l-1)} \circ (\boldsymbol{w}^{(l)T}\boldsymbol{\delta}^{(l)}). \end{aligned} \tag{4.47}$$

In eq. (4.47), $\mathbf{z}'^{(l-1)}$ is a vector with the same shape as $\mathbf{z}^{(l-1)}$ and whose elements are $\partial z_j^{(l-1)}/\partial h_j^{(l-1)}$ for the nodes $\{j\}$ in layer l-1. The Hadamard product in eq. (4.47), $\circ$, is an operation which performs an elementwise multiplication between the elements in $\mathbf{z}'^{(l-1)}$ and our backpropagated matrix product $\mathbf{w}^{(l)T}\boldsymbol{\delta}^{(l)}$, whose shapes are the same. The returned values in $\boldsymbol{\delta}^{(l-1)}$ therefore contains all the gradients $\delta_j^{(l-1)}$ of the nodes $\{j\}$ in layer l-1 from eq. (4.44).

For our example neural network, we can use eq. (4.47) to find $\boldsymbol{\delta}^{(1)}$, if we already have $\boldsymbol{\delta}^{(2)}$, the error of the output layer. The error of the output, which we found in eq. (4.31), was simply the difference between the prediction and the true value

$$\boldsymbol{\delta}^{(l')} = \mathbf{z}^{(l')} - \mathbf{y}. \tag{4.48}$$

Here, l' denotes the output layer and $\mathbf{y}$ is simply the value we want to predict. Once we possess the gradients in the output layer, $\boldsymbol{\delta}^{(l')}$, we can use eq. (4.47) to find the gradients in all the other prior layers of the network.

Knowing how to backpropagate the nodes' gradients in our neural network with matrices, let us also find the matrix equations for the weight and bias gradients.

Beginning with bias gradients, the expression in eq. (4.39) shows us the gradients of the bias parameters are equal to the gradients of the nodes in the same layer, meaning we do not have to make any extra calculations to find the bias' gradients

$$\begin{aligned} \frac{\partial L}{\partial \beta_k^{(l)}} &= \delta_k^{(l)} \implies \\ \Delta \boldsymbol{\beta}^{(l)} &= \boldsymbol{\delta}^{(l)}. \end{aligned} \tag{4.49}$$

We have written the vector holding the gradients as, $\Delta\boldsymbol{\beta}^{(l)}$, which is the matrix holding all the derivatives the loss with respect to all the bias nodes for any layer l. Equation (4.38) shows the weight gradient of the weight connecting the jth preceding node to the kth subsequent note, between layer l-1 and l, is the product of the node in the preceding layer and the error of the node in the subsequent layer. To find the gradient of every single weight, we need to multiply every single preceding node $z_j^{(l-1)}$ with every single subsequent error $\delta_k^{(l)}$. This operation can be written as an outer product between a vector holding the nodes, $\mathbf{z}^{(l-1)}$ and the vector holding the errors, $\boldsymbol{\delta}^{(l)}$:

$$\begin{aligned} \frac{\partial L}{\partial w_{jk}^{(l)}} &= \delta_k^{(l)} z_j^{(l-1)} \rightarrow \\ \Delta \boldsymbol{w}^{(l)} &= \boldsymbol{\delta}^{(l)} \mathbf{z}^{(l-1)T}. \end{aligned} \tag{4.50}$$

Backpropagation for MLP

We dedicate this section to sum up the steps to backpropagating a simple MLP with n layers (including input and output), using matrix notation. Backpropagation is the algorithmic procedure that trains a neural network.

- The first step to backpropagation is forward propagation. Forward propagating the neural network lets us save all the nodes $\{\mathbf{h}^{(l)}, \mathbf{z}^{(l)}\}$, where l indicates which layer the nodes belong to.

- After forward propagating, we can produce the error gradients in the output layer l':

$$\boldsymbol{\delta}^{(l')} = \mathbf{z}(l') - \mathbf{y}$$

- With the error in the output layer, we can calculate and save the error gradients in all the other layers by backpropagating repeatedly:

$$\boldsymbol{\delta}^{(l-1)} = \mathbf{z}'^{(l-1)} \circ (\boldsymbol{w}^{(l)T} \boldsymbol{\delta}^{(l)})$$

- We can then compute the gradients of the weights and biases in all the layers $l \in \{1, ..., n-1\}$ since we now possess the errors and nodes of all layers

$$\Delta\boldsymbol{\beta}^{(l)} = \boldsymbol{\delta}^{(l)}$$
$$\Delta\boldsymbol{w}^{(l)} = \boldsymbol{\delta}^{(l)}\mathbf{z}^{(l-1)T}.$$

- Lastly we update all of our parameters $\{\boldsymbol{w}^{(l)}, \boldsymbol{\beta}^{(l)}\}$ with the gradients $\{\Delta\boldsymbol{w}^{(l)}, \Delta\boldsymbol{\beta}^{(l)}\}$. In section 4.7.4 we present the various ways of doing this.

Backpropagation is slightly more computationally expensive than forward propagating. As we saw in section 4.5.1, forward propagating a neural network of n layers and n nodes per layer results in a time complexity of $O(n^4)$. Backward propagating such a network has a time complexity of $O(n^5)$. This is two orders better than using numerical differentiation for updating the parameters, which has a complexity of $O(n^7)$.

4.7.4 Gradient Descent Variants

Gradient descent has many variants in the field of deep learning. The common denominator between all gradient descent methods is how they use the gradient of a loss function w.r.t. a parameter to change the parameter. In this section we present basic methods such as standard/batch gradient descent, stochastic gradient descent and more advanced variants such as RMSProp and Adam.

Batch Gradient Descent

Updating a neural network starts with us computing the error gradients from a training example $(\mathbf{X}_i, \mathbf{y}_i)$. The computed error gradients for our parameters, $\{\Delta\boldsymbol{w}_i^{(l)}, \Delta\boldsymbol{\beta}_i^{(l)}\}$, corresponds to the single training example $(\mathbf{X}_i, \mathbf{y}_i)$. Say we have a big training set of n training examples: $i = \{1, ..., n\}$. Batch gradient descent performs backward propagation for every single training example, and then takes the mean of our n gradients:

$$\Delta\boldsymbol{W}^{(l)} = \frac{1}{n}\sum_{i=1}^{n}\Delta\boldsymbol{w}_i^{(l)} \tag{4.51}$$

$$\Delta\boldsymbol{B}^{(l)} = \frac{1}{n}\sum_{i=1}^{n}\Delta\boldsymbol{\beta}_i^{(l)}. \tag{4.52}$$

After finding these mean gradients we update the weights and biases

$$\boldsymbol{w}^{(l)} := \boldsymbol{w}^{(l)} - \alpha\Delta\boldsymbol{W}^{(l)} \tag{4.53}$$

$$\boldsymbol{\beta}^{(l)} := \boldsymbol{\beta}^{(l)} - \alpha\Delta\boldsymbol{B}^{(l)}. \tag{4.54}$$

After updating the network like this, we likely have to do it for many more iterations before the minimization of the loss converges (see figure 4.11). Backpropagating the network for an entire training set is often time consuming, and having to do it multiple times makes this method unpopular.

Stochastic Gradient Descent

Stochastic gradient descent, SGD, does what batch gradient descent does, but for small samples of the training set. Instead of calculating the mean of a weight's error gradients over all training examples, this mean is only w.r.t. a random subset of training examples. Say we have $n = 1000$ and a batch size $b = 10$. SGD backpropagates and updates the network for 100 batches b_j of ten random training examples.

$$\Delta \boldsymbol{W}_j^{(l)} = \frac{1}{b} \sum_{i \in b_j} \Delta \boldsymbol{w}_i^{(l)} \tag{4.55}$$

$$\Delta \boldsymbol{B}_j^{(l)} = \frac{1}{b} \sum_{i \in b_j} \Delta \boldsymbol{\beta}_i^{(l)} \tag{4.56}$$

The gradients ($\Delta \boldsymbol{W}_j$, $\Delta \boldsymbol{B}_j$) which we have computed from a small batch of training examples serve as approximations of $\Delta \boldsymbol{W}$ and $\Delta \boldsymbol{B}$, which we can use to update the weights and biases. We can update the parameters, as we did in eq. (4.53) and (4.54), after backpropagating each batch of training examples. How well ($\Delta \boldsymbol{W}_j$, $\Delta \boldsymbol{B}_j$) approximate ($\Delta \boldsymbol{W}$, $\Delta \boldsymbol{B}$) depends on how high the dataset's redundancy is. If the dataset is sufficiently redundant, the increase in update frequency substantially improves the training time. SGD was first presented in Herbert Robbins (1951), decades before neural networks and backward propagation existed.

RMSProp

In spite of its wide use, RMSProp, for Root Mean Square Propagation, is an unpublished method. RMSProp is a method that helps the parameter updates in eq. (4.53) and eq. (4.54) become more consistent. The method scales the gradients of the parameters so that no sudden large change to the parameters take place. The gradients ($\Delta \boldsymbol{W}_j$, $\Delta \boldsymbol{B}_j$) can vary in size for each batch, which can cause a larger, or smaller, change to the parameters than we want, even when the learning rate is constant.

RMSProp scales the gradients with the moving average of their second moment, called *MeanSquare*, $v(\boldsymbol{w}, j)$. The degree to how much this moving average depends on the gradients from the previous and current batch is dependent on a constant

called the *forgetting factor*, γ

$$v(\boldsymbol{w}, j) = \gamma v(\boldsymbol{w}, j-1) + (1-\gamma)(\Delta \boldsymbol{W}_j)^2. \tag{4.57}$$

The index j indicates which batch we are on, but also the timestep of the MeanSquare, $v(\boldsymbol{w}, j)$. For the first timestep, $j = 0$, the MeanSquare is simply the gradients squared: $\Delta \boldsymbol{W}^2_{j=0}$. RMSProp's update rule scales the gradients with the square root of the running mean, v

$$\boldsymbol{w} := \boldsymbol{w} - \frac{\alpha}{\sqrt{v(\boldsymbol{w}, j)}} \Delta \boldsymbol{W}_j. \tag{4.58}$$

The reason why we scale the gradients with a running average of the squared gradients, instead of simply scaling/normalizing the gradients with themselves squared, is because the size of the gradients matter to some degree.

Adam

Adaptive Moment Estimation is an extension of RMSProp optimization. It is often hinted at being the standard optimizer in deep learning and has empirically been shown to perform better than most other popular optimizers in the paper where it was first published (Kingma and Ba 2014). Adam uses the moving average of the second moment of the gradients, like in RMSProp, but also the running average of the gradients themselves.

$$v(\boldsymbol{w}, j) = \gamma_1 v(\boldsymbol{w}, j-1) + (1-\gamma_1)(\Delta \boldsymbol{W}_j)^2 \tag{4.59}$$

$$m(\boldsymbol{w}, j) = \gamma_2 v(\boldsymbol{w}, j-1) + (1-\gamma_2)\Delta \boldsymbol{W}_j \tag{4.60}$$

The running averages are then scaled with their respective forgetting factors, powered by the timestep / batch number

$$\hat{v}(\boldsymbol{w}, j) = \frac{v(\boldsymbol{w}, j)}{(1-\gamma_1^{j+1})}, \tag{4.61}$$

$$\hat{m}(\boldsymbol{w}, j) = \frac{m(\boldsymbol{w}, j)}{(1-\gamma_2^{j+1})}. \tag{4.62}$$

Because the forgetting factors end up with very large exponents in eq. (4.61) and (4.62), the authors suggest setting γ_1 and γ_2 to values between 0.9 and 0.999. The final expression for the parameter update is similar to RMSProp

$$\boldsymbol{w} := \boldsymbol{w} - \alpha \frac{\hat{m}_j}{\sqrt{\hat{v}_j} + \epsilon}, \tag{4.63}$$

where $\hat{m}_j = \hat{m}(\boldsymbol{w}, j)$, same for $\hat{v}_j$, while ϵ is a small scalar which prevents zero division. The Adam optimizer is the sole optimizer used for training neural networks in this thesis.

Chapter 5

Advanced Neural Networks and Regularization

One of the goals in this thesis is to implement different types of neural networks and to see how they compare with traditional forecasting methods. In this chapter we discuss the theory behind specialized neural networks, such as RNNs. For a brief demonstration on how we implement neural networks; see section 6.8.1.

When using neural networks, certain architectures perform better on certain datasets. We have gone in depth regarding the MLP, which is seen as the most basic artificial neural network architecture. Other network architectures build on MLP, but can do a much better job at solving ML problems of a specific nature. Convolutional neural networks (**CNNs**), do well at categorizing image data, while recurrent neural networks (**RNNs**) dominate on problems regarding sequential data such as time series. For this reason we have mostly worked with recurrent neural networks in this thesis.

5.1 Architectures and "The Principle of Locality"

5.1.1 Locality in Images

Convolutional neural networks (CNNs) exploit the principle of locality in image data in order to obtain super-human classification abilities. By locality, we mean that those values that are close to each other, in either an image or a data sequence, form a contextual bond, a kind of information in itself. A CNN does not only look at each pixel in an image uniquely to draw meaning from the image, but at multiple neighbouring pixels at the same time. Exploiting spacial locality in

Figure 5.1: A grey scale image can be represented as a matrix with pixel values between 0 and 255, where 0 represents black and 255 represents white.

this way, the networks is able to draw much more information from the image. A CNN mathematically extracts information from digital images due to the images' numerical form. Looking at grey scale images, we see that images can be represented as matrices on a computer (see figure 5.1).

More explicitly, a CNN transforms, or convolves, neigbouring pixel values in an image into weighted sums. Illustrated in figure 5.2 is an example of a convolution operation done on a section of a grey scale image.

5.1.2 Locality in Time Series

A recurrent neural network (RNN), can be applied on time-series data to utilize the principle of locality in a temporal sense. Just like CNNs use their architecture to look at neighbouring pixels simultatiously, RNNs look at neighbouring values in time/order to predict the next value of a time series. This is beneficial for forecasting time series data, e.g. weather, since yesterday's weather is highly correlated to today's weather. We anticipate that models based on these networks will perform best for the task at hand, which is to forecast time series.

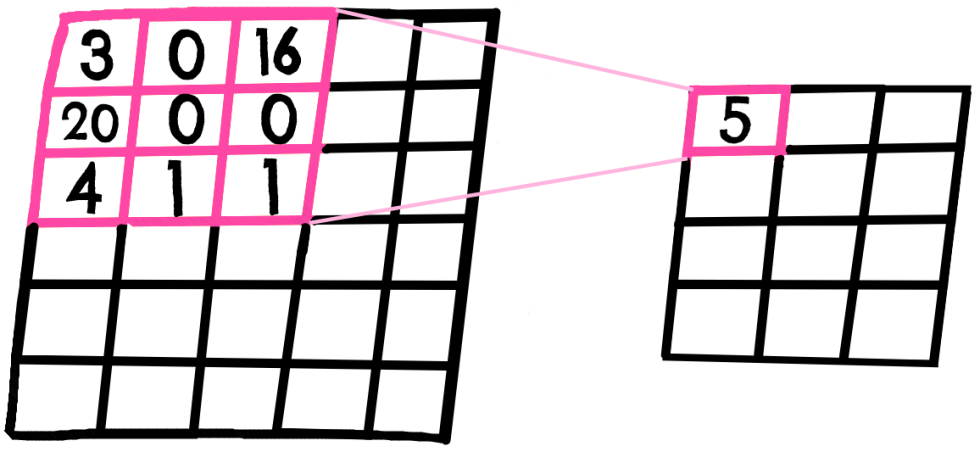

Figure 5.2: Left: A convolutional network's filter (three by three pixels in size here) convolving nine neighbouring pixels in a digital image. Right: The output of the convolution becomes a new image where each cell holds the convolved value. The convolution is a weighted mean between the network's filter and the pixel values. The weights have all been set to 1 in this illustration, giving us the mean.

5.2 Simple RNNs

MLPs typically take an argument $\mathbf{x}$ and maps it to a hidden layer/state $\mathbf{h}$ (Earlier we have used $\mathbf{z}$ to represent a neural net's hidden layer and output layer, but from here on we use $\mathbf{h}$ to represent the hidden layer/state).

$$\mathbf{h} = \varphi(\boldsymbol{w}_h \mathbf{x} + \boldsymbol{\beta}_h) \tag{5.1}$$

RNNs exploit sequential data's locality better than MLPs thanks to their sequential architecture. A simple recurrent network (**SRN**), such as an Elman network can be visualized as a series of T basic MLPs, with one hidden layer, stacked next to each other (see figure 5.3). Each cell in the Elman network is its own little MLP, which processes its input, $\mathbf{x}_t$, together with the hidden state of the previous cell, $\mathbf{h}_{t-1}$, for $t \in \{1, ..., T\}$. This creates a contextual process where the output of each cell in the network depends on the input and output of the previous cells

$$\mathbf{h}_t = \tanh(\boldsymbol{w}_h \mathbf{x}_t + \mathbf{h}_{t-1}\boldsymbol{u} + \boldsymbol{\beta}_h) \tag{5.2}$$
$$\mathbf{z}_t = \tanh(\boldsymbol{w}_z \mathbf{h}_t + \boldsymbol{\beta}_z). \tag{5.3}$$

In eq. (5.2), the input $\mathbf{x}_t$ is multiplied with the weight matrix $\boldsymbol{w}_h$ while the state from the previous RNN cell, $\mathbf{h}_{t-1}$, is multiplied with its own set of weights, $\boldsymbol{u}$. The

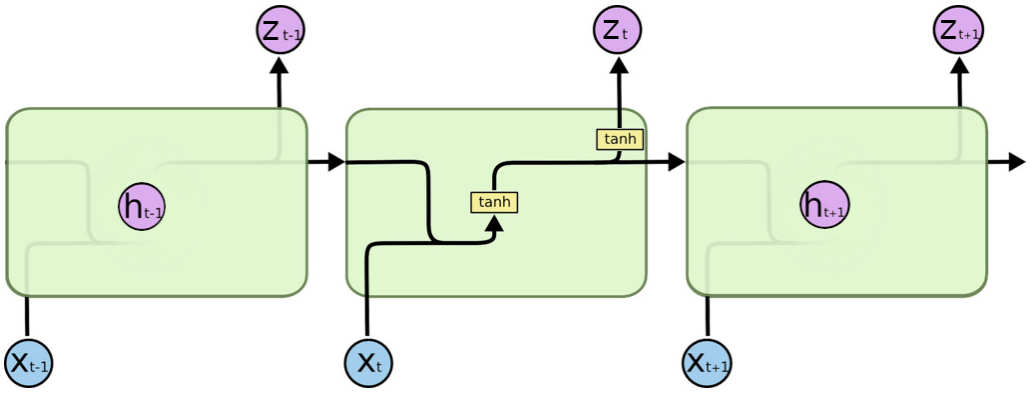

Figure 5.3: Snippet from an Elman network, which is a SRN. The activation function used in this arbitrary Elman network is seen to be *tanh*. Figure based on figure from Karpathy (2015).

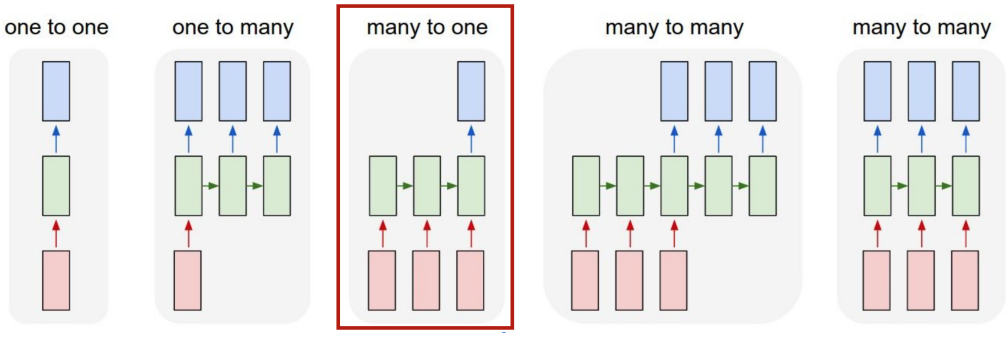

Figure 5.4: The image shows multiple RNN structures where the amount of "cells" vary. The red blocks are input layers $\mathbf{x}_t$, the green are the hidden states $\mathbf{h}_t$, while the blue are the output $\mathbf{z}_t$. In our regression problem, we use a "many to one" model. This figure is based on a figure from Karpathy (2015)

forward propagated values are then summed together to form the hidden state in the current state, $\mathbf{h}_t$. Eq. (5.3) propagates the hidden state to the output my multiplying it with the weights $\boldsymbol{w}_z$. The weights and biases are the same for every cell in an RNN. It is basically the same neural network inferred on itself recurrently.

For prediction problems, the output of an RNN's last cell, $\mathbf{z}_T$, is usually the only output that matters. When only computing the output of one RNN cell, we call it a "many to one" network (see figure 5.4).

Training a recurrent neural network is no different to training an MLP. The cells of an RNN functions very much like layers in an MLP when we unpack it

$$\mathbf{z}_t = \varphi(\boldsymbol{w}_z \mathbf{h}_t + \beta_z) \tag{5.4}$$

$$= \varphi(\boldsymbol{w}_z(\mathbf{x}_t \boldsymbol{w}_h + \mathbf{h}_{t-1}\boldsymbol{u} + \beta_h) + \beta_z). \tag{5.5}$$

After unpacking the network's cells as layers, we simply compute the output

$\mathbf{z}_{t=T}$ for a training example, compute the error of the output and then recurrently backpropagate w.r.t. the weights and biases.

5.3 LSTMs

Simple RNNs with many cells risk problems like the vanishing gradient problem if the network's activation function is of a saturating nature (sigmoids). Long short-term memory (**LSTM**) is a type of recurrent neural network, published in Hochreiter and Schmidhuber (1997), shown to be resistant to both the vanishing and exploding gradient problem. Due to this resistant nature, an LSTM can consist of thousands of cells/timesteps, allowing previous information from far back in time to affect the present outcome of a prediction.

One cell in an LSTM network is more complex than the cells in a simple Elman network. The original LSTM variants possess an "input" and "output" gate. The input gate is a set of weights which regulate the flow of new values into the cell. The output gate controls how much of the new values in the cell is used to calculate the cell's output. A couple of years after the LSTM was published, new variants with "forget gates" became the standard for LSTMs. The forget gate controls the degree to which a value of the LSTM cell remains and was first published in **forgetgate**.

The three gates are computed in a similar fashion, each with their own set of weights $\boldsymbol{w}$ and $\boldsymbol{u}$, where $\boldsymbol{w}$ forward propagates the input features while $\boldsymbol{u}$ propagates the previous hidden state vector $\mathbf{h}_{t-1}$

$$\mathbf{i}_t = \sigma(\boldsymbol{w}_i \mathbf{x}_t + \boldsymbol{u}_i \mathbf{h}_{t-1} + \boldsymbol{\beta}_i) \tag{5.6}$$
$$\mathbf{o}_t = \sigma(\boldsymbol{w}_o \mathbf{x}_t + \boldsymbol{u}_o \mathbf{h}_{t-1} + \boldsymbol{\beta}_o) \tag{5.7}$$
$$\mathbf{f}_t = \sigma(\boldsymbol{w}_f \mathbf{x}_t + \boldsymbol{u}_f \mathbf{h}_{t-1} + \boldsymbol{\beta}_f). \tag{5.8}$$

With these gates, we compute the cell state and the hidden state for the cell, $\mathbf{c}_t$ and $\mathbf{h}_t$

$$\mathbf{c}_t = \mathbf{f}_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tanh(\boldsymbol{w}_c \mathbf{x}_t + \boldsymbol{u}_c \mathbf{h}_{t-1} + \boldsymbol{\beta}_c) \tag{5.9}$$
$$\mathbf{h}_t = \mathbf{o}_t \circ \tanh(\mathbf{c}_t). \tag{5.10}$$

In contrast with an Elman cell, the LSTM cell forwards two outputs to the subsequent cell in the network, both $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$. The initial values of these output vectors are zero: $\mathbf{c}_0 = \mathbf{h}_0 = 0$. The $\circ$ operator is the Hadamard product, while σ is the sigmoid activation function. We have provided an illustration of an LSTM cell in figure 5.5. In this thesis, the most frequent type of neural networks contain LSTMs.

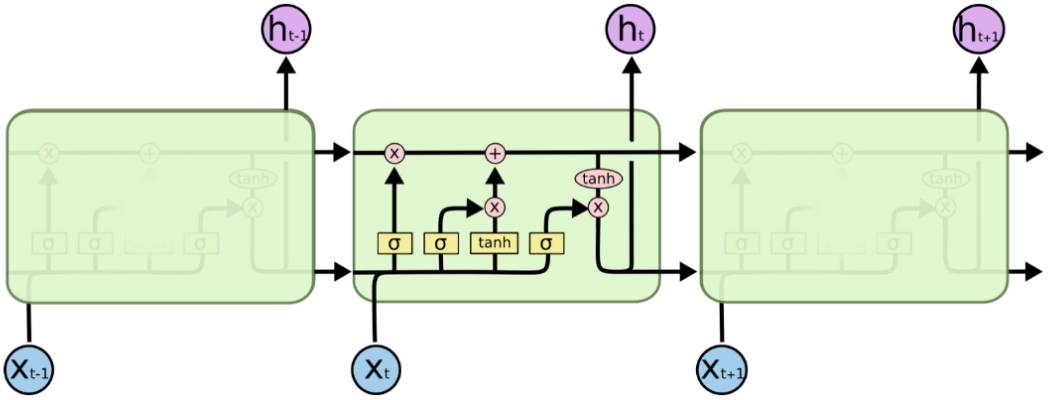

Figure 5.5: The gigure shows how the inputs $\mathbf{x}_t$, $\mathbf{h}_{t-1}$ and $\mathbf{c}_{t-1}$ are propagated through the various gates in an LSTM cell, resulting in the output of $\mathbf{c}_t$ and $\mathbf{h}_t$. The $\times$ symbols represent Hadamard products, elementwise multiplication.

5.4 Sparse Encoding and Embeddings

Input layers are usually filled with nodes containing numerical values such as stock prices and pressure values or other quantities, but what do we do if we also want to tell the network which season it is?

5.4.1 One Hot Encoding

A simple way to make categorical data into numbers that make sense is to give each category its own scalar index: 0 = winter, 1 = spring, 2 = summer and 3 = fall. We then simply put these number into an input node in the input layer. The problem with this solution is that all four seasons share the same weights. The only thing that distinguishes them is their index. The two most differing seasons in our solution is winter (0) and fall (3), which does not make sense since summer should be furthest from winter. We can conclude that simply representing categories as scalar numbers is an inadequate scheme.

It's popular to represent categorical data points as vectors instead of scalars to meaningfully distinguish between categorical values in a neural network. We start by creating a vector as long as the number of categories we have, which is four in this example. We then fill the vector with zeros, except for the in the index that corresponds with the scalar index we have designated to the category. This way of representing categorical data is called "one hot vector encoding" and is visualized in figure 5.6.

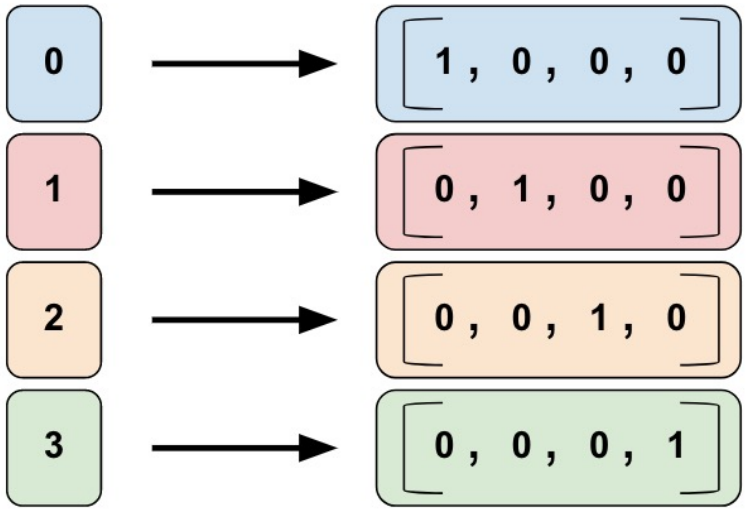

Figure 5.6: In order properly distinguish between categorical data, we can represent them as "one hot vectors". In a one hot encoding scheme, a categorical datapoint of the nth category is represented as a zero vector, where the nth index is set to 1. Figure taken from Tensorflow (n.d.).

5.4.2 Embeddings

One hot vectors, which represent categorical data in a neural network's input layer, is typically connected to a set of weights in the form of an embedding. An embedding can be pictured as a simple weight matrix, which we multiply with a one hot vector. The embedding weight matrix is randomly initialized and its elements are optimized just like any other parameter in a neural network. It has a number of rows equal to the number of categories. The product between a one hot vector of the nth category and the embedding weight matrix is simply then nth row of the embedding weight matrix (see figure 5.7).

We are not limited to only feeding the network one type of categorical data. In this thesis we have experimented with feeding the network data about weekday, month, and time of day, all at the same time. To tell a neural network what weekday it is, we need a one hot vector of length seven and to tell it which month it is, we need a vector of length 12. The vector for Monday would look like this: $[1, 0, 0, 0, 0, 0, 0]$ and the vector for January would look like this: $[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$. When feeding a network two types of categorical data, such as which weekday and month it is, we concatenate the one hot vectors:

$$x_{Monday\ in\ January} = \begin{bmatrix} 1\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{bmatrix}$$

This gives us a sparse vector of length 19, meaning our embedding weight matrix needs to have 19 rows. Figure 5.8 illustrates how a sparse input vector such as ours interacts with the embedding to form three hidden nodes, parallell to an MLP.

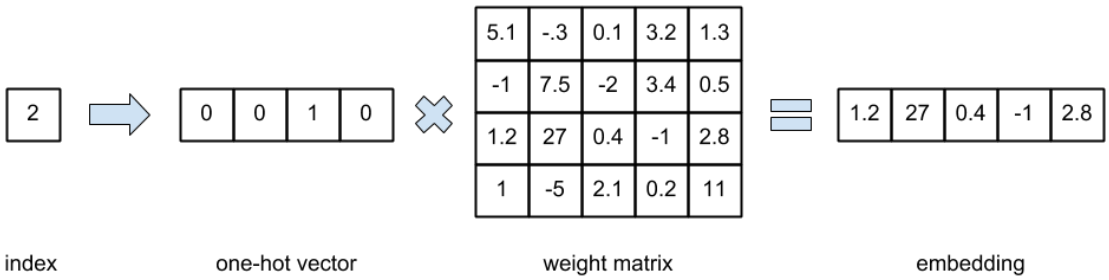

Figure 5.7: The figure shows the pipeline of a categorical datapoint from input to forward propagation. We begin by giving each categorical data value a scalar index corresponding with its category. Secondly, we turn said index into a one hot vector. The one hot vector functions like an input layer, which we multiply with the weight matrix in order to receive our hidden nodes (here dubbed embedding).

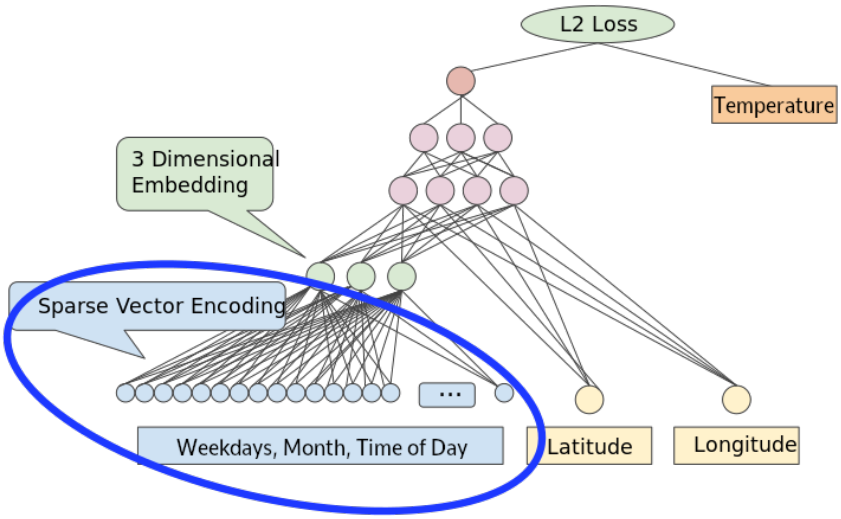

Figure 5.8: The figure shows a feed forward neural network, or MLP, which is connected to an embedding. The embedding propagates the sparse input vector of originally categorical data into three hidden nodes, here called "three dimensional embedding". The hidden nodes from the embedding and the non-categorical input, "longitude" and "latitude", are forward propagated to the subsequent hidden layer. From what we see in the image, the neural network is attempting to predict a place's temperature by looking at its geographical position and time of year. The image is based on a figure from Google (2019).

There are no fixed rules regarding what to do with hidden nodes spawned from embeddings. The three nodes born from the embedding in figure 5.8 are concatenated with the input layer, containing the "latitude" and "longitude" inputs. Another alternative for what to do with the three embedding nodes in the figure could have been to simply sum them with the subsequent hidden layer, or concatenate them to it. The best alternative is like to be found through testing.

In our thesis we have incorporated embeddings together with LSTMs in order to improve our predictions of differential pressure. The edge we obtain in machine learning when using categorical depends on what we wish to predict. Since the pressure values we are analyzing are dependent on human behavioral patterns, categorical data regarding holidays have also been used in order to improve our models' accuracy.

5.5 Regularization

Neural networks, with their ability to mimic any function, are at risk of overfitting on the data set which it is trained on (see section 3.4 for overfitting). Many advanced methods have been designed to counter act this overfitting. In modern neural networks, layer normalization, batch normalization, group normalization, Dropout and other methods have newly been devised to make neural networks perform "better", for a lack of a better word. These methods have in common that they indirectly cause a neural network to generalize better. In this thesis we have looked at classical regularization methods such as the $L2$ norm, and newer ones such as Dropout and found that the latter performed better for our problem.

5.5.1 *L2* Norm Regularization

As the name somewhat suggests, $L2$ norm regularization takes the $L2$ norm of all the weight matrices in the network, multiplies it with a small scalar and appends it to the cost function. Since the cost function is minimized during training, the network will penalize the weights for adding complexity to the model, reducing the overfitting. The $L2$ norm of an $N \times M$ weight matrix is simply the sum of all its weights squared

$$L_2(\boldsymbol{w}) = \sum_{j}^{N} \sum_{i}^{M} w_{ij}^2. \tag{5.11}$$

In statistics, a problem solved using $L2$ norm regularization is called a ridge regression, while $L1$ norm regularization, which looks at the absolute value of the weights instead of their squared value, is called LASSO regression. In machine learning

problems where we use stochastic gradient descent, we call $L2$ regularization for "weight decay".

5.5.2 Dropout

New regularization methods such as Dropout (Srivastava et al. 2014), have little to do with mathematical theorems and are perhaps the reason why some AI researchers call their field "alchemy". The highly successfull method is responsible for reducing overfitting in neural networks and has proven quite useful in this thesis. Dropout picks a random subset of nodes in a neural network, while it is training on a batch, and turns them off. That is to say, all values these nodes propagate forward become zero. The inhibition of a subset of nodes forces the other nodes to learn more efficiently and results in much less overfitting. Dropout is the main form of regularization used for our models.

Part II

Methodology

Chapter 6

Data Pipeline and Implementation

Before we start implementing models, we need to have our data in order. The first goal of our thesis is to make a data pipeline which turns our asynchronous and aperiodical data into uniform time series.

In this chapter we discuss different aspects of the pipeline, from start to finish. The pipeline sorts, extracts, resamples, scales and feeds the data to our models. It consists of different steps, some that we only have to do once, and some which we repeat every time we want to train a new model. We will present each step our data goes through, both graphically and with snippets of code, before ending up in the inputs of one of our models.

6.1 Sorting by Name

To make sense of one year worth of measurements made by thousands of sensors, we organised the measurements by the tag they belonged to. Each measurement a tag makes includes its name, making it easy for us to divide our data into individual files. In section 2.4 we showed this snippet of the original data file:

```
"2018-03-09 22:22:00","XXX_PT705.vY",747.59,192,192,4
"2018-03-09 22:23:00","XXX_PT705.vY",733.39,192,192,4
"2018-03-09 22:24:00","XXX_PT705.vY",735.43,192,192,4
"2018-03-09 22:25:00","XXX_PT705.vY",739.50,192,192,4
"2018-03-09 22:26:54","XXX_PT705.vY",766.21,192,192,4
"2018-03-09 22:27:12","XXX_PT705.vY",739.57,192,192,4
"2018-03-09 22:28:01","XXX_PT705.vY",750.72,192,4
"2018-03-09 22:29:01","XXX_PT705.vY",746.80,192,192
```

6.2 Reading Tags into Arrays

Tags are the main focus of this thesis. When we predict future pressure values in the heating system, it is measurements made by a specific tag we are predicting. The tag

$$“XXX\_PDT2002.vY”$$

(which from here on will be called PDT2002) is of special interest to the company who has given us the dataset. Most of our models have been developed and trained on PDT2002, however, our methods can be inferred on any other tag in the dataset. In order to predict tag PDT2002's future measurements, we feed our models its previous measurements (univariate forecasting). This means we have to be able to effortlessly read tags from our district files into arrays in our programs. The Numpy and Pandas packages in Python solve this problem and lets us quickly extract the tags we want.

```
#test.py
import pandas as pd
array = pd.read_pickle('path_to_tag/tag.pickle')
array.head() #show first values in dataframe
```

```
 $python3 test.py
 >>
                            Value
Date
2017-06-28 16:02:57  248.870667
2017-06-28 16:02:58  251.745056
2017-06-28 16:03:59  253.693481
2017-06-28 16:04:59  251.926849
2017-06-28 16:05:59  250.785965
```

After reading tags into arrays, we can start processing and plotting the data. In figure 6.1 you can see a plot of the measurements made by some tags over a small time interval.

6.3 Resampling

As you can see in figure 6.1, a tag's measurements happen at aperiodical timesteps. Feeding inconsistently spaced values in time into a forecasting model is going to make our model confused. The model will treat measurements that are 2 minutes away from each other no different from measurements that are 5 hours away from each other, resulting in meaningless predictions. To solve this, we make our measurements

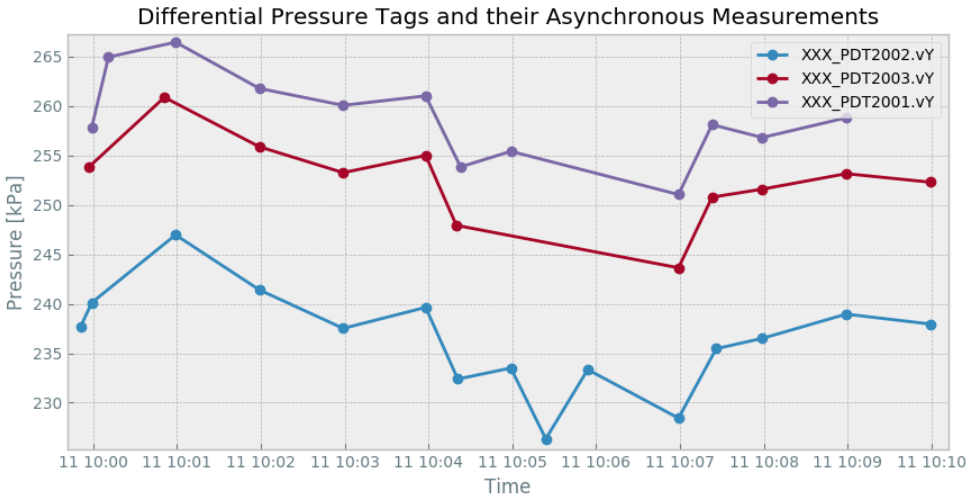

Figure 6.1: The figure shows a time interval of differential pressure measurements made by tags in the heating system. Notice that the measurements are aperiodical and asynchronous.

equally spaced in time. The patterns our model finds will then be patterns that are consistent with the normal passage of time. To give our data a constant timestep, we use a linear interpolation scheme which maps all measurements to rounded timesteps (eg. 00:00, 00:01, 00:02). Our code has been implemented to allow us to dynamically choose the size of our timestep.

```
#test2.py
time_interval = 60 #timestep size (seconds)
array = pd.read_pickle('path_to_tag/tag.pickle')
array = array.resample('\%ds'\% time_interval).mean()
print(array)
```

```
$python3 test2.py
>>
                              Value
Date
2017-06-28 16:02:00   250.307861
2017-06-28 16:03:00   253.693481
2017-06-28 16:04:00   251.926849
2017-06-28 16:05:00   250.785965
2017-06-28 16:06:00   248.639206
```

In figure 6.2 and the listing above, you can see how this interpolation modifies the data features in figure 6.1 to have a uniform timestep, but in the figure we see that some timesteps are vacant of any values.

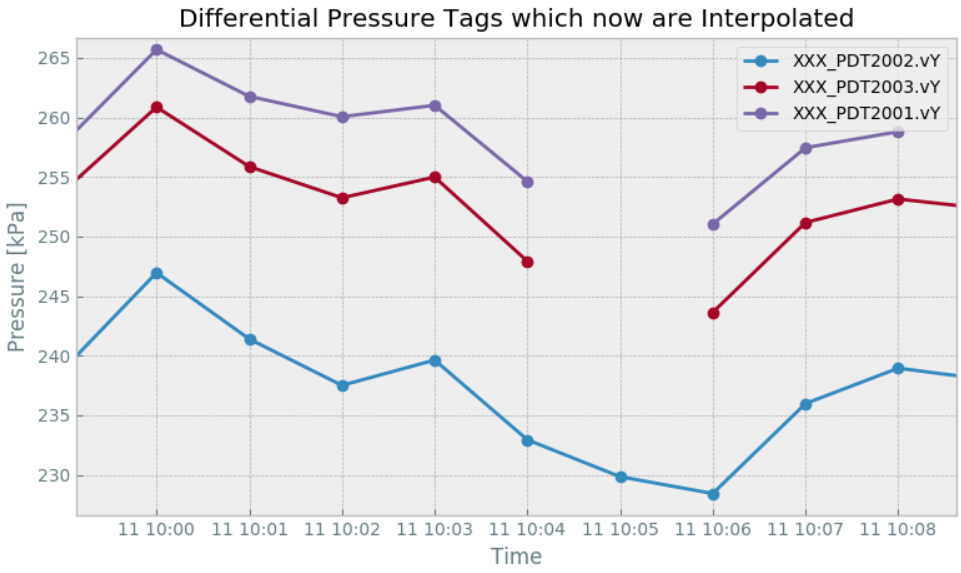

Figure 6.2: The figure shows an interpolated version of the tags in figure 6.1. The interpolation maps our data to rounded timestamps. In this figure we have the the timestep to one minute, which maps all values to their closest rounded minute. The vacancies in the data were there to begin with, but are a problem for our analysis nonetheless.

6.4 Filling Vacancies in the Data

In figure 6.2 we can observe timesteps where no measurements exist (the 5 minute mark). Our models are going to struggle with analyzing such a holey data sequence, since they need an actual input $\mathbf{x}_t$ at all timesteps t. We therefore want to fill these vacant positions with values. As mentioned in section 2.4, our sensor tags are event driven. The sensors make measurements every second and only saves a new value to its array when there is a significant change from the last measurement. From this we can presume that wherever there is a vacancy in a tag's measurement history, the best approximate value to fill the void time interval with is the last measurement the tag made. This is a method called **forward fill**. Applying forward fill to the tags in figure 6.2 results in i voidless dataset which we can use train machine learning models on.

```
array = pd.read_pickle('path_to_tag/tag.pickle')
array = array.resample('\%ds'\% time_interval)
#fill vacancies with preceding value
array = array.fillna(method='ffill')
```

In figure 6.3 is the forward filled version of the tag arrays from figure 6.2.

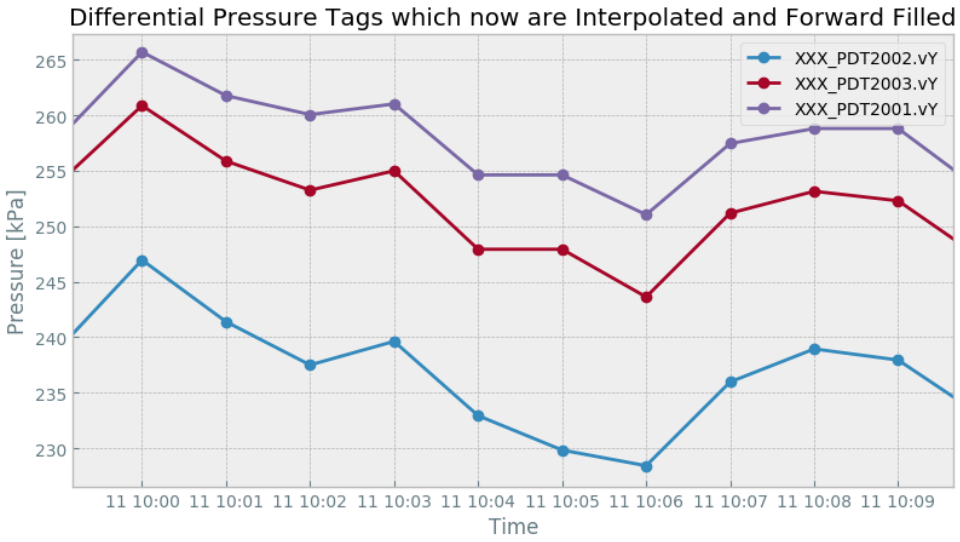

Figure 6.3: The figure shows an interpolated and forward filled version of the tags originally depicted in figure 6.1. Now that all our data is synchronized and periodical, we can start using it to train ML models.

6.5 Feature Scaling

In machine learning problems, it is normal to scale you data in order make it easier for the models to find patterns in them. Each feature we intend to feed or predict in our neural net will be standardized. Standardization is a way to scale a dataset, which sets the data's variance to 1, and its mean to 0

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - \mu_x}{\sigma_x}, \quad \hat{\mathbf{y}} = \frac{\mathbf{y} - \mu_y}{\sigma_y}, \tag{6.1}$$

where σ is the standard deviation in the data, and μ is the mean. Standarization (Yann et al. 1998) greatly helps gradient-based learning, which neural networks depend on. The scaling, however, results in our neural network having to predict unitless labels $\hat{\mathbf{y}}$. The output predictions of our neural network, $\hat{\mathbf{z}}$, will therefore also be unitless and so will have to be transformed back to their original units

$$\mathbf{y} = \hat{\mathbf{y}}\sigma_y + \mu_y \implies \tag{6.2}$$

$$\mathbf{z} = \hat{\mathbf{z}}\sigma_y + \mu_y. \tag{6.3}$$

The scaling is quite easy to implement, below is an snippet from our code that is performing standardization on all input features:

```
means=[] #we save the mean and standard dev.
stds=[]  #- of each feature array
for i in dims: #dims is the number of input features
```

```
    mean = Arrays[i].mean()
    std = Arrays[i].std()
    Arrays[i] -= mean
    Arrays[i] /= std
    means.append(mean)
    stds.append(std)
```

Once a feature is scaled, we can feed it into a model. If the model is multivariate, we scale all the features individually, as the code above illustrates. After getting our predictions, $\hat{\mathbf{z}}$, we can scale them back to real units (kPa) with the help of eq. (6.3), letting us to calculate the physical error:

```
[...] #post regression
#rescale predictions to physical units
predictions = predictions*stds[label_index] + means[label_index]
residuals = (predictions - y_test).abs() #test errors
MAE = residuals.mean() #mean absolute error in physical units
```

6.6 Creating Data Batches

The last step of the data pipeline is the data's transition into a model's input. Since we have resampled, and forward filled each feature, we know that each value is equally spaced in time. To refer to one measurement, we write x_t. To refer to a measurement which comes after x_t, we write x_{t+k}, where k is the number of timesteps into the future from when x_t took place. In time series forecasting, we want to predict future values x_{t+k}, by using older values like x_t, x_{t-1}, x_{t-2}, etc. as inputs simultaneously.

6.6.1 Samples

To create training examples, we first need to define far into the future we want to see and how many preceding timesteps we want to use as inputs simultaneously. Say we have a feature, $\mathbf{x}$, and we want to predict which value comes after x_t, e.g. x_{t+1} ($k = 1$). As inputs for the prediction, we take a set of n measurements from time t and earlier: $\{x_t, x_{t-1}, ..., x_{t-(n-1)}\}$. Using a series previous values to predict the ones that come after is called "autoregression". To evaluate the prediction, $\hat{x}_{t+k}$, we compare it to the label, x_{t+k}. The input and the label forms an input-output pair which make up any supervised learning dataset. We also call these pairs for training examples or simply "samples"

$$\mathbf{x}_{n=3,t} = (\{x_t,\ x_{t-1}\ x_{t-2}\},\ x_{t+1}). \tag{6.4}$$

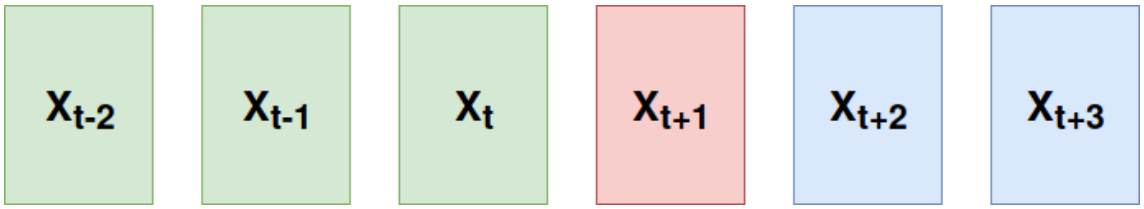

Figure 6.4: The diagram is a visualization of a feature's values over time. Marked green are the inputs for an input sample $\mathbf{x}_{3,t}$, while in red we find the label we want to predict.

In eq. (6.4) is a sample consisting of the $n = 3$ preceding time steps from time t, and a label that is $k = 1$ time steps into the future. Figure 6.4 is visual representation of this sample, where the input is marked green and the label is marked red.

6.6.2 Batches

In the previous subsection we explained how to make a single sample from a feature array, and we chose to call such a sample $\mathbf{x}_{n,t}$, where n are the number of inputs and t is timestep of the most present input. One sample is a very tiny piece of data and is not going to be enough to train a model. We are going to need samples. We briefly touched on batches in section 4.7.4. Batches are collections of samples, usually in groups to the power of two (e.g. 32, 64, 128, etc.).

If you look at the sample $\mathbf{x}_{3,t}$ from figure 6.4, you might notice how trivial it is to create a new sample, $\mathbf{x}_{3,t+1}$. To create more samples we simply move the inputs and label of the previous sample one timestep forward. More specifically, what we do is take every term in $\mathbf{x}_{3,t} = (\{x_t,\ x_{t-1}\ x_{t-2}\},\ x_{t+1})$ and we iterate them one step forward in time: $\mathbf{x}_{3,t+1} = (\{x_{t+1},\ x_t\ x_{t-1}\},\ x_{t+2})$. Repeating this process can be visualized as a filter moving along the timesteps, collecting samples. Once the sample filter reaches the edge of our feature array, all possible samples have been created, which we can begin feeding into our models as batches. In figure 6.5 we have visualized the filtering-like process of creating samples.

In our code, we separate the input and output pairs of the sample

$$\mathbf{x}_{n,t} = \{x_t,\ ...,\ x_{t-(n-1)}\},\ y_{n,t} = x_{t+k}.$$

Here is a snippet of code inspired by our own, however, this snippet is not our actual code. The actual code for this stage is a bit uglier and much less intuitive as a consequence of wanting it to run faster.

```
n = 30 #number of preceding timesteps we want in each sample
k = 1  #number of timesteps into the future we predict
```

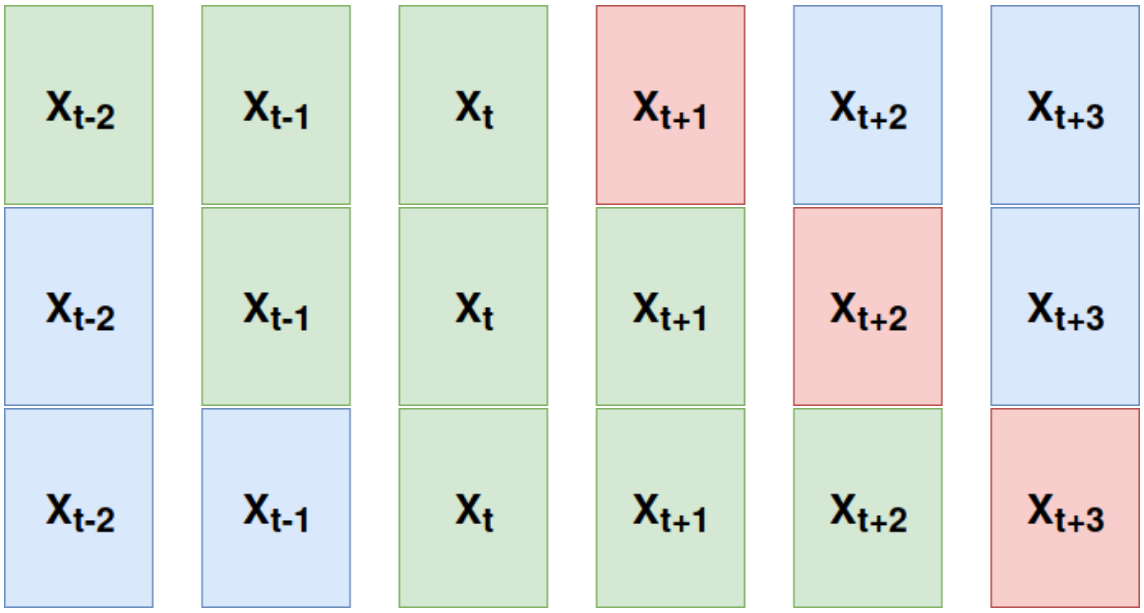

Figure 6.5: The first sequence is the same as the one in figure 6.4 and shows the inputs and label of sample $\mathbf{x}_{3,t}$, where $\mathbf{x}_{3,t} = (\{x_t,\ x_{t-1}\ x_{t-2}\},\ x_{t+1})$. The two following sequences illustrate the next two samples we can collect, $\mathbf{x}_{3,t+1}$ and $\mathbf{x}_{3,t+2}$. To create each of these samples, we simply move the inputs and label to be their neighbouring values, one timestep into the future. As a reminder, the green values are the inputs and the red value is the output of the sample.

```
for t in timesteps[n:-k]:
    r = range(t-(n-1),t) #number of preceding values as inputs
    r.append(t+k)
    dataX[t] = array[r[:-1]] #input samples
    datay[t] = array[r[-1]]  #labels
```

After we have sampled the data, we divide them into a training, test and validation set before training a model. More on optimization in section 8.3.

```
test_percentage = 20 #20% of the samples go to the test set
test_cursor = int((1-test_percentage/100)*datax.shape[0])
#test cursor is the index at the 80% mark of the samples

testx = dataX[test_cursor:]
testy = datay[test_cursor:]
trainX = dataX[:test_cursor]
testy= datay[:test_cursor]
```

6.7 Differencing the Data

In time series forecasting, predicting a feature's future value is not always the goal. Sometimes, predicting the **difference** between the present value and future value results in better predictions. Doing this requires us to perform "differencing" on our features. To differentiate our inputs and labels, we take our feature vector $\mathbf{x}$, of length n, and simply subtract the $n-k$ last features with the the $n-k$ earliest features. The resulting feature vector is similar to the original one, except that each value is the difference between its original value and the value k timesteps in the past

$$\Delta\mathbf{x} = \begin{bmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_n \end{bmatrix} - \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-k} \end{bmatrix}. \tag{6.5}$$

Once we have a differenced feature vector, we can begin creating samples and batches as we have described in figure 6.5.

When differencing we want to predict $\Delta x_{t+k} = x_{t+k} - x_t$, where k is the number of timesteps into the future. Usually when predicting a differenced time series, our inputs are also differenced. An expression for a regression model which predicts the difference between the present feature value (corresponding to timestep t) and the future feature value can be written as $f(\Delta x_t, \Delta x_{t-1}, ...) = \Delta\hat{x}_{t+k}$, where $\Delta\hat{x}_{t+k} + x_t = \hat{x}_{t+k}$ is the model's prediction of x_{t+k}.

In this thesis, we have tried using differencing to boost our predictions' accuracy. Differencing a feature emphasizes the short term relationships in the data series. There also happens to sometimes exist long term relationships inside time series, also known as Long Memory or Long-Range dependencies. Geophysical time series, such as river flow data, has been shown, to contain long term relationships (Hurst 1951). It seems reasonable to presume that the Heating System also contains similar long term relationships which we want to exploit. Econometrician Granger and economics professor Joyeux wrote about differencing in, Granger and Joyeux (1980), where they noted that: "Some econometricians were reluctant to this technique (differencing), believing that they may be losing information, by zapping out the low frequency components.". Presumably, differencing removes long term relationships in time series.

6.8 Input Data and Model Implementation

This section is dedicated to present the final destination of the data pipeline as well as deliver som insight into how we make our models.

6.8.1 Implementing a Small Neural Network

After the data pipeline has created all our samples and we have divided the dataset into a test and training set, we can begin making our models. To make neural network models, we have used the Python library; Keras. For linear AR models, we use the scikit-learn library. Let us walk through how we make and train neural network models. The first step is to import everything we need from Keras.

```
from tensorflow.keras.layers import Dense, Input
from tensorflow.keras.models import Model
import tensorflow as tf
```

Second, we define our input layer. Let us assume our data samples consist of the 20 preceding values of a single feature.

```
n = 20 #samples contain 20 of the preceding values
input_layer = Input(shape=([20,1]))
```

If we are to make a multilayer perceptron for example, then we need to add at least one hidden layer and an output layer:

```
n = 20
h = 128 #number of hidden nodes
input_layer = Input(shape=([n,1]))
hidden_layer = Dense(h,kernel_initializer =
            'glorot_normal',activation='ReLU')(input_layer)
output_layer = Dense(1,kernel_initializer =
            'glorot_normal',activation='linear')(hidden_layer)
```

The word “Dense” is used to describe a simple feed forward propagation. Notice how Keras connects the layers together because we pass the previous layer as an argument to the subsequent one. There are other ways of building a neural network in Keras, this way is called “the functional API”. We are not done yet, however, what remains is as intuitive to implement as what has already been seen. After making a model with the “Model” class, we train the model by deciding which optimizer to use and things like batch size and validation set size:

```
validation_percentage = 5
batch_size = 64 #how many samples to train on per update
model = Model(inputs=input_layer,outputs=output_layer)
optimizer = tf.keras.optimizers.Adam()
model.compile(loss='mean_squared_error', optimizer=
                    optimizer,metrics=['mse','mae'])

model.fit(trainX, trainy,
    validation_split=validation_percentage/100,
            batch_size=batch_size)
```

Chapter 7

Feature Reduction and Selection

In the previous chapter, we talked about the data pipeline and how it processes our data before sending it to a model's input layer. Before data is sent down the pipeline, however, most of it is removed by heuristics described in this chapter. Additionally, we discuss how we pick fitting features for multiregression models.

7.1 Manual Feature Reduction

Feature is a word used to describe an array of input data, such as the measurements from a tag. A hard nut to crack in this thesis is how to cleverly choose which features to feed into our models to predict another. We possess thousands of tags, 4-5 types of weather features, and multiple artificially created features which potentially could help boost our predictions. Naively attempting to feed the entire dataset into our model to predict PDT2002 makes it difficult for a model to distinguish good or bad features. This section is dedicated to explain how we have reduced the amount of tags in our dataset by using non-analytical heuristics.

7.1.1 Local Models

The first step to selecting relevant tag features for our multivariate models, is to find those with something in common with the label tag. By label tag, we mean the tag we are trying to predict the behavior of (PDT2002). Thousands of tags measure pressure, the unit of our tag of interest, yet these tags behave very differently. The difference in the tags' behavior is clearly because they are placed in entirely different locations inside of Oslo's Heating System. By following something along the lines of the principle of locality (see section 5.1), we assume tags lying near each other

are likely to have a valuable correlation between their measurements. The features' spacial information is therefore, presumably, exploitable. To exploit tags' locality it, we compare their district code. As a reminder: Each sensor has a name that starts with its area code, followed by the unit it measures:

"`XXX_PDT3401.vY`".

As mentioned earlier in this chapter, I have redacted/switched the area codes with the letters "XXX". Altogether, there are tens of thousands of tags in our system. By only focusing on tags that belong to a common district when predicting pressure measurements of a tag in said area, we manage to reduce the amount of candidate input tags from thousands to hundreds.

7.1.2 Removing Lazy Tags

Some of the tags in our dataset are almost or completely inactive. Intuitively, a tag that rarely makes a measurement is not a helpful input feature that will help our models learn to predict PDT2002. Removing "lazy tags" is done by setting a lower threshold value of 25'000. If the amount of measurements a sensor has made for an entire year is lower than the threshold, we disregard it (such a sensor makes $\lessapprox 2.5$ measurements per day on average). Removing such lazy tags potentially relieves us of 9/10 tags, depending on the district the tag we are predicting belongs to. Specifically, the area of PDT2002 possesses 976 sensors of which 834 are lazy. The remaining 142 tags are the only ones we deem eligible to become input features.

To sum up our feature reduction steps:

- We have reduced thousands of potential input features to hundreds by only training models with tags from our label tag's (PDT2002) district.
- We further reduced the remaining number of features by removing the most inactive of them.

7.2 Feature Selection

In this section, we will explain the less heuristic methods we have used to choose the optimal input tags for a specific response variable / label tag, using filter feature selection methods. **Feature selection** is a crucial field within **data science** that tackles the problem of finding optimal input features for your model. Feature selection methods can be divided into three categories:

- **Filters**

  Filters are methods that measure relevance between an input feature and a label feature. This relevancy is not clearly defined, but popular quantities to measure are: covariance and Pearson's correlation coefficient. If an input feature scores high on these tests, it shows that there is a correlation or dependence between it and the label. We keep the input features that score high on these tests and dump the ones that do not.

- **Wrappers**

  Wrappers are methods that turn you optimization problem into a search algorithm. Some wrappers, such as the Recursive Feature Elimination algorithm (Guyon et al. 2002), trains a model on the entire feature set, then removes features of lesser relevance. Other wrappers train multiple models on a lot of subsets of your features. Each model's prediction error ends up functioning as a score which determines the best model. Wrappers usually give the best result when optimizing through feature selection, but they cost much more computational resources than other methods.

- **Embeddings**

  Embeddings are characterized as methods that perform feature selection during the model's training. Embedding methods penalize those input features that are characterized as noise, zeroing them out with a penalizing weight. LASSO regression (Santosa and Symes 1986), is perhaps the most commonly known example of an embedding method. In neural networks however, an embedding is typically a trainable feed forward layer which transforms combinations of categorical/binary values to real values. See section 5.4 for more uses of embeddings.

7.3 Filtering Methods

We have looked at two filtering methods and will in this section discuss each one's characteristics. The methods are all used in a univariate feature selection scheme, meaning we investigate the response variable's dependence on one other feature at a time. The features that score highest are used as input variable to multivariate regression models, which we compare to models that are univariate.

7.3.1 Covariance and Correlation

To see if our label tag / response variable PDT2002, Y, is dependent on a feature tag, e.g. X, we can calculate their covariance. Covariance, expressed in eq. (7.1),

can be interpreted as a measure of how similarly two variables vary

$$\sigma_{XY} = \frac{1}{N}\sum_{t=1}^{N}(X_t - \mu_X)(Y_t - \mu_Y). \tag{7.1}$$

In eq. (7.1), t denotes the timestep of the variables, μ is the average of each variable and N is the total number of timesteps.

If we scale the covariance between the response and the feature variable with their standard deviation, we get a measure of their linear correlation, the so-called Pearson correlation coefficient.

$$\rho_{X,Y} = \frac{\mathtt{cov}(X,Y)}{\sigma(X)\sigma(Y)}. \tag{7.2}$$

The Pearson correlation coefficient, $\rho_{X,Y}$, can tell us how linearly dependent X and Y are. The coefficient's value will vary between -1 and 1. A high negative or positive value implies a strong linear dependency between the variables. Whereas 0 correlation translates to a lack of any linear dependence. Note that the variables may be non-linearly dependent anyway, making this method suboptimal for selecting features in non-linear regression problems. This method has performed quite well for us nonetheless.

In figure 7.1 we have plotted a heatmap of a small chunk of something called the **correlation matrix**, for all the active features close to PDT2002. The code needed to calculate the correlation, ρ, between all the features exists in the Pandas library of Python, which is what we used. For n features $X_1, ..., X_n$, the correlation matrix is an $n \times n$ matrix where the (i, j) index is the Pearson correlation coefficient $\rho(X_i, X_j)$ from eq. (7.2), between feature i and j. The variables with the highest absolute correlation are used in our multivariate models to predict the response variable PDT2002.

7.3.2 Mutual Information and Entropy

Since our neural network models are able to find non-linear relationships between inputs and label, a high mutual information score, **MI**, is a presumably a viable criteria for adding features to our model. The reason for this is that MI represents both linear and non-linear correlation between two variables.

Mutual information is a quantity that measures how much one variable tells us about another. In statistical terms, it is defined as how much the uncertainty of one variable is reduced, given knowledge of the other. By uncertainty we mean the entropy, which is a measure expressed in eq. (7.3).

$$S(X) = -\sum_{i} P_X(i) \log P_X(i) \tag{7.3}$$

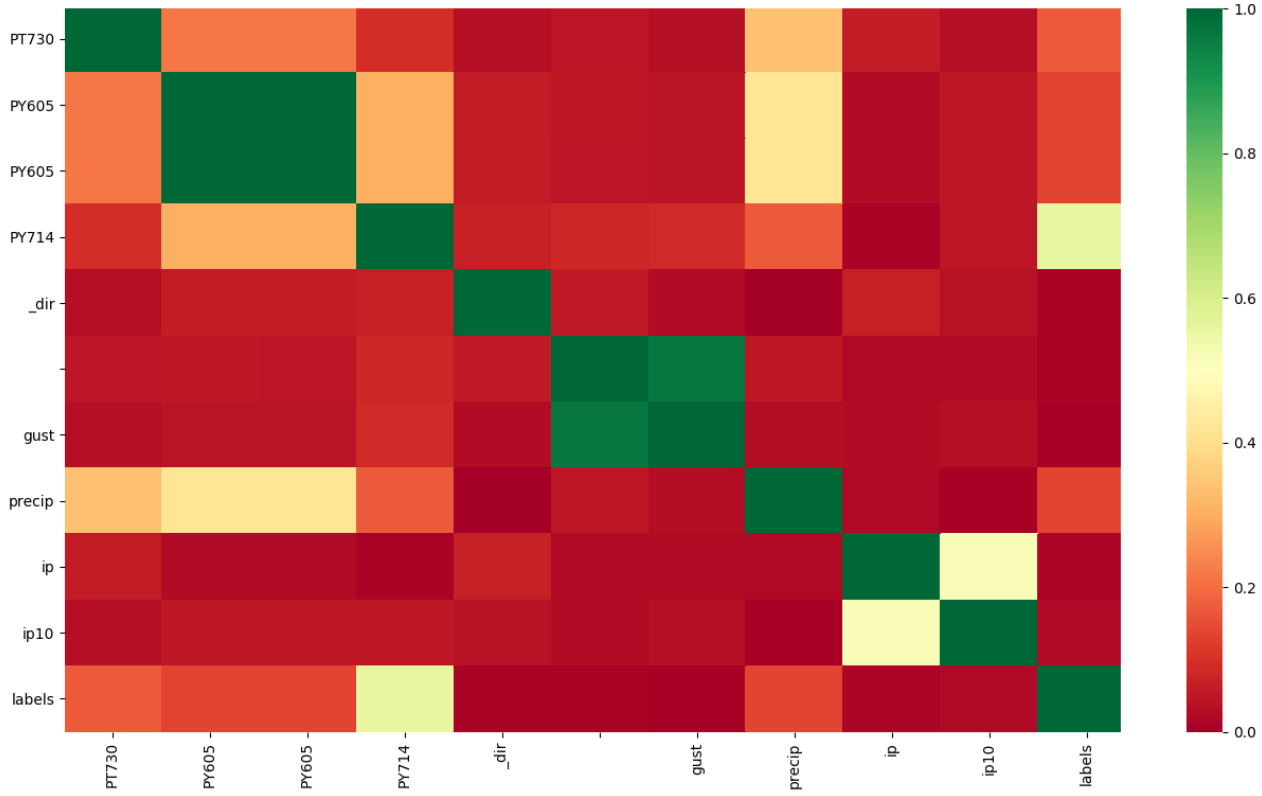

Figure 7.1: The correlation matrix shows us the degree to which of our variables are linearly correlated. I've plotted absolute correlation since we don't need to distinguish between negative and positive correlation. The diagonal is green because the correlation between one variable and itself is equal to 1.

In our case $P_X(i)$ is the value $P_X(X_i)$, where P_X is the probability distribution function of X, X_i is the i'th value of feature X and $P_X(X_i)$ is the marginal probability of X_i. A probability distribution is a bit problematic when dealing with a real valued problem since our real valued features mostly have unique values. The probability distributions of the features are therefore uniform, which is unhelpful. MI was actually for categorical variables, who tend to have a finite number of classes. The typical solution to making a meaningful probability distribution for real values is to use binnings. The binned entropy is expressed similarly as eq. (7.3) in eq. (7.4)

$$S(X) \approx S^*(X) = -\sum_x P_X^*(x) \log P_X^*(x), \tag{7.4}$$

where the sum function iterates over the bins, x, of X's binned probability distribution; P_X^*. $P_X^*(x)$ is the marginal probability for the x'th bin in X. The next expression needed for calculating the MI is the conditional entropy, $S(X|Y)$

$$S(X|Y) = \sum_y P_Y(y) \left[-\sum_x P_{X|Y}(x|y) \log \left(P_{X|Y}(x|y) \right) \right]. \tag{7.5}$$

In eq. (7.5), $P_{X|Y}(x|y)$ is the probability of x given y where x and y are values/bins of X and Y. With this we can write the Mutual Information as

$$I(X;Y) = S(X) - S(X|Y). \tag{7.6}$$

Notice that I goes to zero if $S(X|Y) = S(X)$ in eq. (7.6), which is synonymous with the variables being statistically independent/non-correlated.

Chapter 8

Optimization

Creating predictive models is a multi-step process in which we need to design models, optimize, run benchmarks and repeat. In this chapter, we discuss the complicacies of creating models of value by focusing on their features, hyperparameters and architecture.

8.1 Input Features

Linear regression models, MLPs and LSTMs are the three types of models which we have used to perform time series forecasting. All three have been used to perform univariate analysis, which in this thesis means: A regression which trains a model on multiple inputs contiguous in time from a **single** feature, in order to predict the feature's value k timesteps into the future (go back to figure 6.5 to get a visualization of the inputs and prediction labels of a time series). Only LSTMs have been used for making multivariate models, where a model takes **multiple** sequences of inputs, each sequence being from a unique feature. Since LSTMs have an architecture which discerns which timestep the input belongs to, we assume they are less likely to be overwhelmed by multiple input features.

8.1.1 Numerical Input Features

Most of the candidate input features we have are the measurements of tags such as PDT2002. Besides tags from the heating system, we also have weather measurement data features of Oslo taken from the Norwegian Meteorological Institute. These features measure wind speed and precipitation. Lastly, we created a numerical encoding for time data without using embeddings. Realizing features such as "time

of day" or "week day" can be represented as sine and cosine functions with periods equal to 24 hours or 7 days, we can use these sine and cosine functions as inputs to our models. This did sadly not give us any improvements. The encoding method is presented in an online article at Wyk (2018).

The last section in the previous chapter discusses how to find relevant input features for multivariate regression by using linear correlation and MI filters. Knowing the most correlated features to our label, PDT2002, still leaves the question: How many features should we simultaneously use as inputs to boost prediction accuracy? The answer is found by simply trying with different amounts. The number of features to input into our multivariate models is effectively a hyperparameter, which we have quite a lot of in in our models.

8.1.2 Categorical Input Features

In section 5.4 we presented embeddings and explained how they allow us to feed categorical data into our network. The categorical data we have used in this thesis is mainly temporal. That is to say, we have experimented with how our network responds to being told which week day it is, which month it is and what time of day it is. We have also created a categorical feature which tells the network whether or not if there is a large event in Oslo at the moment, such a a soccer game or a public holiday. The categorical features we created were: time of day, week day, week number, month and holiday.

8.2 Hyperparameters

Regarding models, we have primarily discussed neural networks with their parameters; weights and biases. These are parameters which we optimize during the training/learning process. Hyperparameters are parameters which we set before the training begins. This includes the number of nodes in a neural network's layer, the number of layers, which activation function to use, the size of the timestep in our data, the amount of timesteps into the future we want to predict and so on.

Our problem possesses many hyperparameters, making it difficult for us to optimize a each model we make. The choice of value for our hyperparameters is crucial in optimization and can lead to a huge boon in model accuracy, but since hyperparameters are set before the training process, optimizing hyperparameters is a time costly affair. Below are some bullet points presenting and discussing some of our hyperparameters and the scope inside which we plan to vary them:

- **Timestep Size**

  The original data is technically sampled at 1 Hz. This is unnecessarily frequent. We can downsample and interpolate and forward fill the data and still avoid aliasing seasonal patterns. Decreasing the resolution of our data into a constant timestep grants us a very easy way of defining how far into the future we want to predict. We will set our timestep to fixed values between one and 20 minutes to see how it affect predictions.

- **Future Vision**

  How far into the future do we want to predict? From a naive perspective, as long as possible, but from a practical perspective, as far into the future as is needed to respond to the foreseen event. Modern Heating Systems are water based and Oslo's require many minutes, e.g. 30, to properly respond to an energy increase from a heat plant. We will therefore try to predict PDT2002's behavior from one to 40 minutes into the future. Predicting the differential pressure tag's behavior at multiple steps will give a contiguous picture of what might happen in the future, which is much more insightful than only knowing what might happen in exactly 30 minutes.

- **Input Sample**

  The input to our model can be a large amount of preceding values from our features' time series. We get to define whether to look one minute or many hours back in time, when trying to estimate the a future data point in the series. In a series where the timestep size is 5 minutes, ten timesteps translates to the last 50 minutes of feature values. We will experiment with using input samples of 3-20 timesteps for our models, regardless of timestep size.

- **Network Architecture**

  In the case of Artificial Neural Networks, multiple of hyperparameters are introduced to a regression problem. Such as a model's layer type (e.g. feed forward or recurrent), the number of layers, the number of nodes in each layer, the type of activation function behind each layer, whether to use regularization or not, which loss function or optimizer to use and how many epochs to train.

  - **Models and Layers**

    Our MPLs all have four layers. Our recurrent neural network models are either LSTMs followed by a feed forward layer, or two feed forward layers.

  - **Nodes and Layers**

    To make things easy for us, we use 128 hidden nodes in each feed forward layer, and in each of the weight matrices in the LSTM gates.

  - **Activation Functions**

    We have extensively tested both ReLU and sigmoid in our models. The nodes connected to the output node is activated by the linear activation function.

  - **Regularization**

    We have used $L2$, Dropout and sometimes neither.

  - **Loss Function and Optimizer**

    The Mean Squared Error is the sole loss function used to train networks. The Adam optimizer is likewise the only optimizer we have used. The Adam optimizer has three initial parameters: learning rate, β_1 and β_2, which all can be treated as hyperparameters.

- **Epochs**

  An epoch passes every time a model has been trained on the entire training set. A model can typically train on the training set multiple times before it overfits. For this reason we have let our models train for ten epochs, and saved the network with the best validation accuracy each time.

- **Differencing**

  Differencing the data can be thought of as a binary hyperparameter which we can turn on or off whenever we please. We have extensively tested it and found that it has its pros and cons prediction.

8.3 Evaluation

To improve our models, we minimize their training error, $L(\mathbf{X}_{train}, \mathbf{y}_{train})$. More important than the training error is the model's performance on data it has not trained on. In supervised learning it is customary to have both a validation set and a test set which the model is not allowed to be fitted on. Both datasets are used to evaluate the model, but each in their own way.

8.3.1 Test Error

The test error, $L(\mathbf{X}_{test}, \mathbf{y}_{test})$, is the most robust metric for evaluating our model and corresponds, in our thesis, to the MSE error of our models' predictions on the test data. The test error is calculated **after** the training process and is usually computationally costly, but much less so than the training itself. By taking the square root of the test MSE gives us the average prediction error, which we easily can convert to physical units [kPa] by following the steps in section 6.5. The average physical test error is the result we use when comparing how accurate our models' predictions are.

8.3.2 Validation Error

To achieve the minimum test error possible, which is essentially the same as avoiding overfitting, the validation set was made. The validation set, which we touched on in section 4.7.1, is typically smaller than the test error. While 75% of the samples in PDT2002 go into the training set and the test set holds 20% of the samples, the remaining 5% defines the validation set $\{\mathbf{X}_{val}, \mathbf{y}_{val}\}$. Just like a mini test set, we do not train our model on the validation set. Instead of using the validation set to evaluate the model after training, we evaluate on it **during** training.

By using the validation set to evaluate our model between epochs, we can stop training as the network beings overfitting. Since the validation set is so small, the evaluation does ideally not slow down the training/fitting process. The validation error, $L(\mathbf{X}_{val}, \mathbf{y}_{val})$, like the test error, shows us how good a model's predictions are on data it has not yet trained on. As each training epoch passes, the training error and validation error diminish, but eventually the validation error will start increasing (see figure 8.1). This means overfitting is taking place.

Instead of stopping training when the validation error starts increasing, we train each model for ten epochs and save the weights and biases whenever the validation error decreases. The reason why we do not immediately stop the training when the validation error starts increasing is because it might decrease a couple of epochs later.

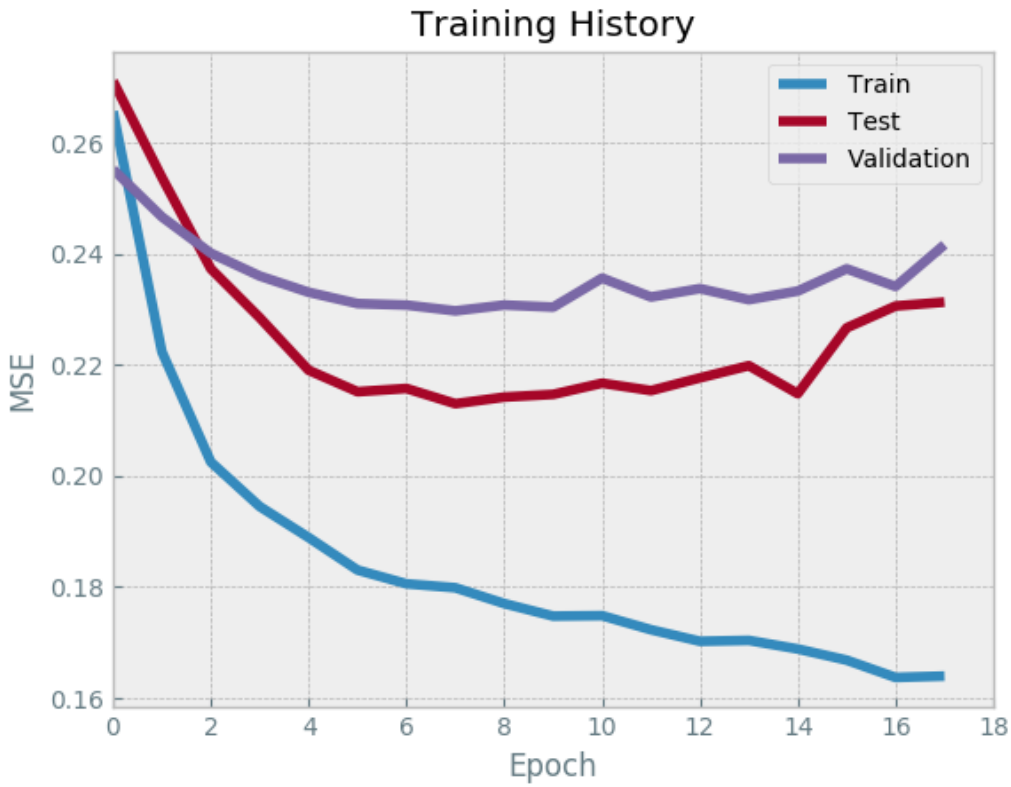

Figure 8.1: The figure shows a graph of the training, test and validation MSE error for a neural network model. As the training progresses, the model overfits. In epoch seven, the model seems to have the smallest validation and test error, making it an ideal time to stop training. We do not calculate the test error after each epoch, that is what the validation set is for.

Part III

Comparative Analysis and Results

Chapter 9

Univariate Analysis Results and Hyperparameter Optimization

The first regression models we made were neural networks such as MLPs, LSTMs with a feed forward layer and LSTMs with two feed forward layers. The LSTMs with two feed forward layers and the MLPs were free to use either ReLU or sigmoid activation functions in their first feed forward layer, while the activation function connected to the output has to be the linear activation function. In this chapter we present the results of our univariate models and which hyperparameter values worked best.

9.1 LSTMs

Especially many models were trained in the first rounds of optimization, because we did not yet know which hyperparameter values were optimal for our problem. During the univariate model optimization, we mainly dealt with different LSTMs while tweaking almost every single hyperparameter. Although the univariate models are far from our best models, training them gave us valuable insight into the hyperparameters which resulted in minimum prediction error.

As described in the hyperparameter section, 8.2, we have many types of hyperparameters: timestep size (for the data), input sample size, differencing the data and adding an extra feed forward layer after the LSTM, resulting in the possibility for adding an activation function - a hyperparameter in itself. We have looked at six timestep sizes and ten different sizes of input samples. The amount of unique combinations of hyperparameters in our univariate problem are $6 * 10 * 2 * 3 = 360$. Making a model for all of these configurations is called "grid search", which is what

we have done. Additionally, each of these 360 models had to be trained up to nine times in order to make them predict different distances into the future.

9.1.1 Univariate LSTM Results

Below, in table 9.1, is a compact representation of the most successful models in our grid search analysis. Each row in the table describes an LSTM model with a unique set of hyperparameters, each of which outperformed douzens of other LSTMs with a similar architecture, but different hyperparameter settings.

Table 9.1: The table shows the hyperparameters belonging to the models with the lowest prediction errors. Each model has a set data resolution (timestep size) as well as a set distance into the future which it has been trained to predict.

Future[min]	Timestep[s]	Input Sample	Differenced	MAE [kPa]	Activation
2	60	25	Yes	5.47	ReLU
3	180	30	Yes	7.36	ReLU
6	120	30	No	13.03	None
10	60	15	No	19.67	None
15	180	20	No	26.04	ReLU
20	60	20	No	30.84	Sigmoid
30	300	20	No	35.89	None
35	60	20	No	37.77	None
40	60	25	No	38.05	None

The column ”Future“ shows how far into the future the model was trained to predict, while ”MAE“ is the mean absolute test error of the model. An activation of ”None“ means that the LSTM has one feed forward layer with a linear activation function instead of two feed forward layers. The models with two feed forward layer following the LSTM layer had either ReLU or sigmoid as the activation function in the first feed forward layer. The input sample size denotes how many LSTM cells are in the network, as well as the length of the input data sequence.

9.1.2 Hyperparameter Insights

Conclusions we can draw from table 9.1 is that differencing is good for short term predictions, but bad for long term predictions. There also seems to be a trend in the amount of timesteps to use as inputs. All our top models prefer between 15 and 30 timesteps when predicting, meaning the models need to look far back in time in

order to make better predictions. This implies that PDT2002's feature vector has long term dependencies and explains why differencing ruins long term predictions.

In addition to many timesteps being a preferred hyperparameter setting, so it seems that small timesteps lowers prediction error. This is not unexpected, since downsampling our data to have larger timesteps between each value will cause some information to disappear. We also risk aliasing periodic patterns in our features if we have make too large timesteps. For this reason, the preferred timestep for our other analyses will be as 60 and 120 seconds. Setting the timestep size to 30 seconds did not improve our models and only slowed the learning process.

A lot of hyperparameters regarding the network had little no effect. Initial learning rate, hidden nodes in the hidden layers, batch size were all initially optimal, with the learning rate set to 0.001, the hidden nodes to 128 and the batch size to 64. We did notice small improvements when halving the learning rate to 0.0005 and so we kept it halved.

9.1.3 Benchmarking with Persistence Models

Before benchmarking our findings with linear regression models we want to present an even more basic baseline as a benchmark. In order to find out if our forecasting models are any good, a standard way of benchmarking in time series forecasting is to compare your regression model with a persistence model. A persistence model is a model which assumes that no change is going to happen to the feature $\mathbf{x}$ over time. As an example, a persistence weather model assumes the weather tomorrow is the same as today. The persistence model, in other words, predicts that the future feature value of $\mathbf{x}$, x_{t+k}, is equal to the most recent value of $\mathbf{x}$ it is given as an input (e.g. x_t)

$$\begin{aligned} f(x_t) &= \hat{x}_{t+k} \\ &= x_t, \end{aligned}$$

where k is the number of timesteps into the future you want to predict. The predictions made by a persistence model are bad, but not terrible. Their predictions are accurate for features with low variance, since they always guess that the variable will not vary. In figure 9.1 we have visualized a persistence model trying to predict what PDT2002's feature values will be, one timestep into the future.

The mean absolute test error of a persistence model varies with how far it predicts. Below, in table 9.2, we can see how our best univariate models' results from table 9.1 and the persistence model's results compare to each other. The results show that our models beat the persistence model, but this is not something we should brag about.

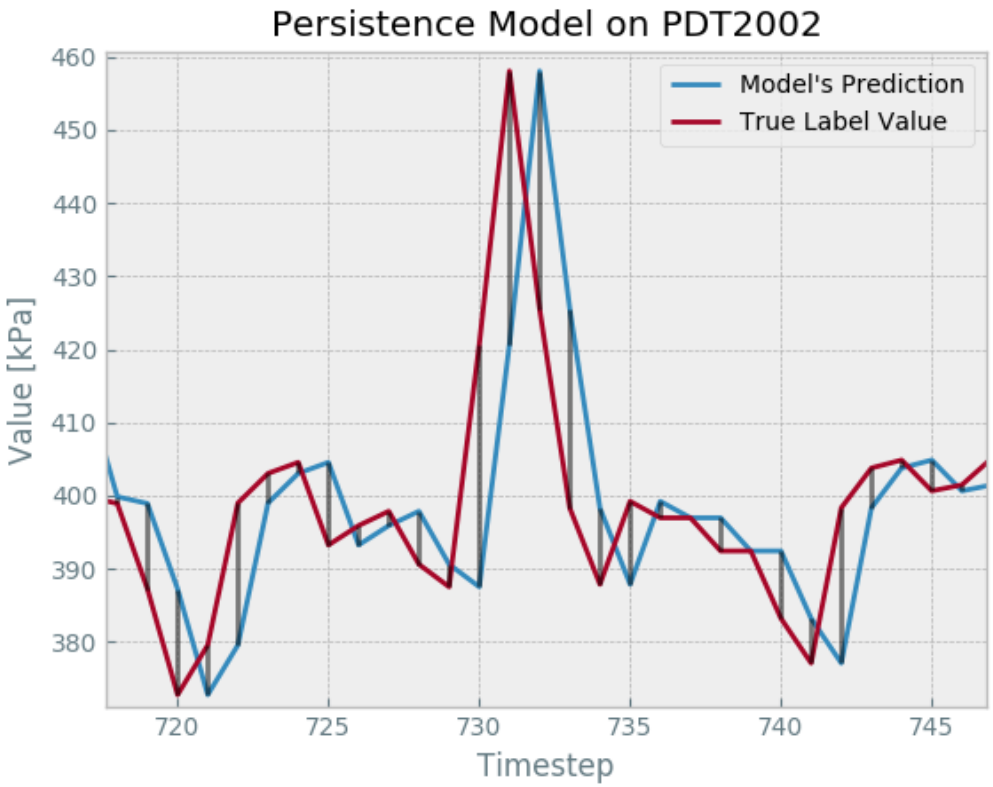

Figure 9.1: The figure shows a persistence model next to the PDT2002 feature. Since the model knows the value corresponding to the current timestep of the feature, it simply predicts the next value to be the current one. At some points, this turns out to be accurate, but sudden changes in the feature creates large errors which are visualized as see-through residuals.

Table 9.2: Above is a comparison of our best univariate LSTM model, vs. the persistence model's mean prediction error on PDT2002's test set.

Future[min]	LSTMs MAE [kPa]	Persistence MAE [kPa]
2	**5.47**	6.50
3	**7.36**	8.77
6	**13.03**	15.19
10	**19.67**	22.37
15	**26.04**	29.51
20	**30.84**	34.96
30	**35.89**	42.46
35	**37.77**	44.97
40	**38.05**	46.79

Because we feel like comparing nine separate LSTM models with with the persistence model is a bit unfair, so we trained a final univariate LSTM model (with a single feed forward layer after the LSTM layer) to give a visual representation of how the models fare against each other. The hyperparameters are inspired by what we learned from our first analysis: timestep size of 60 seconds, 20 timesteps as input and no differencing. The comparison between the LSTM and the persistence model is visualized in figure 9.2.

9.2 Linear Prediction and MLPs

Other benchmark models we can compare our LSTMs to are linear autoregression models and multilayer perceptrons. In time series forecasting, linear regression models used on samples of previous feature values to predict future values is called Linear Prediction. Linear Prediction models have a bit fewer hyperparameters than a neural network, only timestep size and input sample size. We presumed the same hyperparameters that benefited LSTMs also benefits Linear models and MLPs. We therefore trained models with input samples of 20-30 timesteps and timestep sizes of 60-120 seconds.

9.2.1 MLPs

Multilayer perceptrons like LSTMs are neural networks and can therefore be customized to a greater extent than Linear Prediction models. We chose to make the MLP networks 4 layers long, with 2 feed forward layers and 128 hidden nodes in the two hidden layers. Just like our MLPs, the one in figure 9.3 illustrates a 4 layer

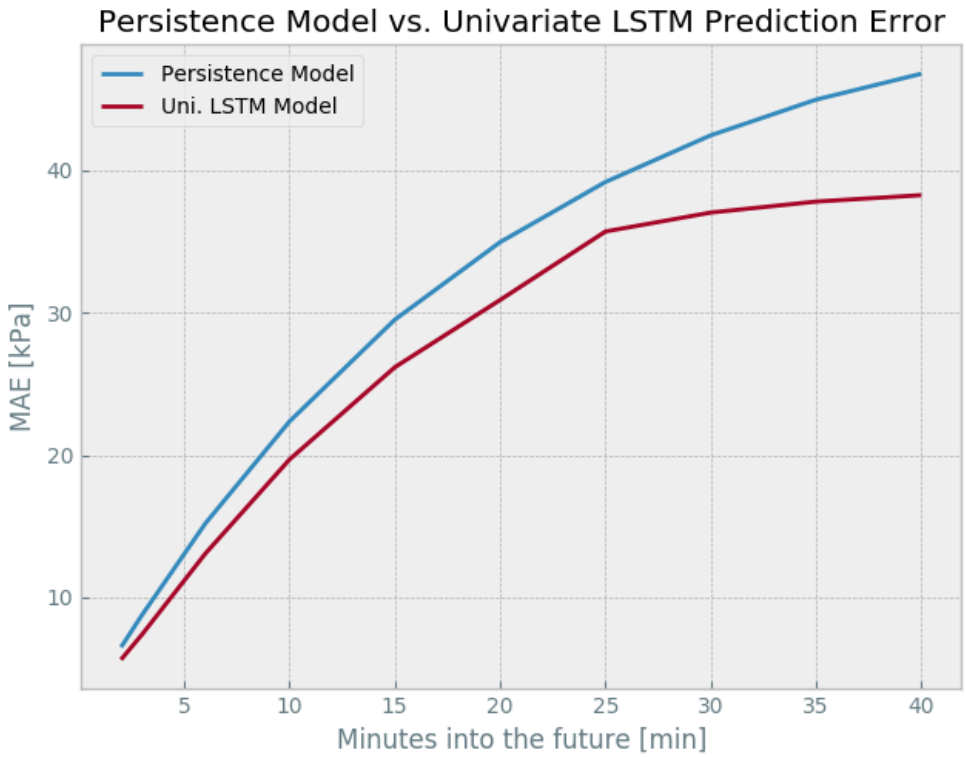

Figure 9.2: In blue is the average absolute test error of the persistence model and in red is the same error of the LSTM. As we try to look further into the future, the errors seem to increase somewhat logarithmic.

MLP, except that it has way fewer hidden nodes. Half of the networks we trained had ReLU activation in their first hidden layer and the other half used sigmoid. The input nodes, corresponding to the input sample size, varied from 20 to 30 inputs. The predictions of the MLPs are slightly worse than the LSTM, which is expected of non-RNN networks who do not exploit a sequence's locality. In figure 9.4 is a plot of the prediction error of the best MLP, which has a timestep of 60 seconds, input sample of 25 timesteps and a sigmoid activation function, next to the latest LSTM we trained in the previous section from figure 9.2.

9.2.2 Linear Prediction Results

Our linear autoregression models performed increasingly better on long term predictions (20+ minutes forward in time) when the timestep size of the data got bigger (300-600 seconds), but this in turn increased the short term prediction error. Table 9.3 contains the mean test errors of the two linear regression models which came out on top. In figure 9.5 we have visualized the prediction error of the linear model

Table 9.3: The table lists the test error of two Linear Prediction models. They both have an input sample size of 30 timesteps, but possess different timestep sizes. The model trained on data with smaller timesteps is better at short term predictions, while the model trained at 300 second timesteps is better at long term predictions (20 minutes and longer).

Future[min]	**MAE [kPa], (Timestep = 60s)**	**MAE [kPa], (Timestep = 300s)**
2	5.69	–
3	7.58	–
5	**11.45**	11.53
6	13.32	–
10	**19.67**	20.18
15	**26.04**	26.73
20	31.71	**31.31**
30	37.25	**36.42**
35	38.73	**37.71**
40	39.69	**38.49**

trained on 60 second timesteps, in order to visually compare it to our univariate LSTM model. Linear models seem more competent that MLPs when it comes to predicting both short and long term, but fall short of univariate LSTMs.

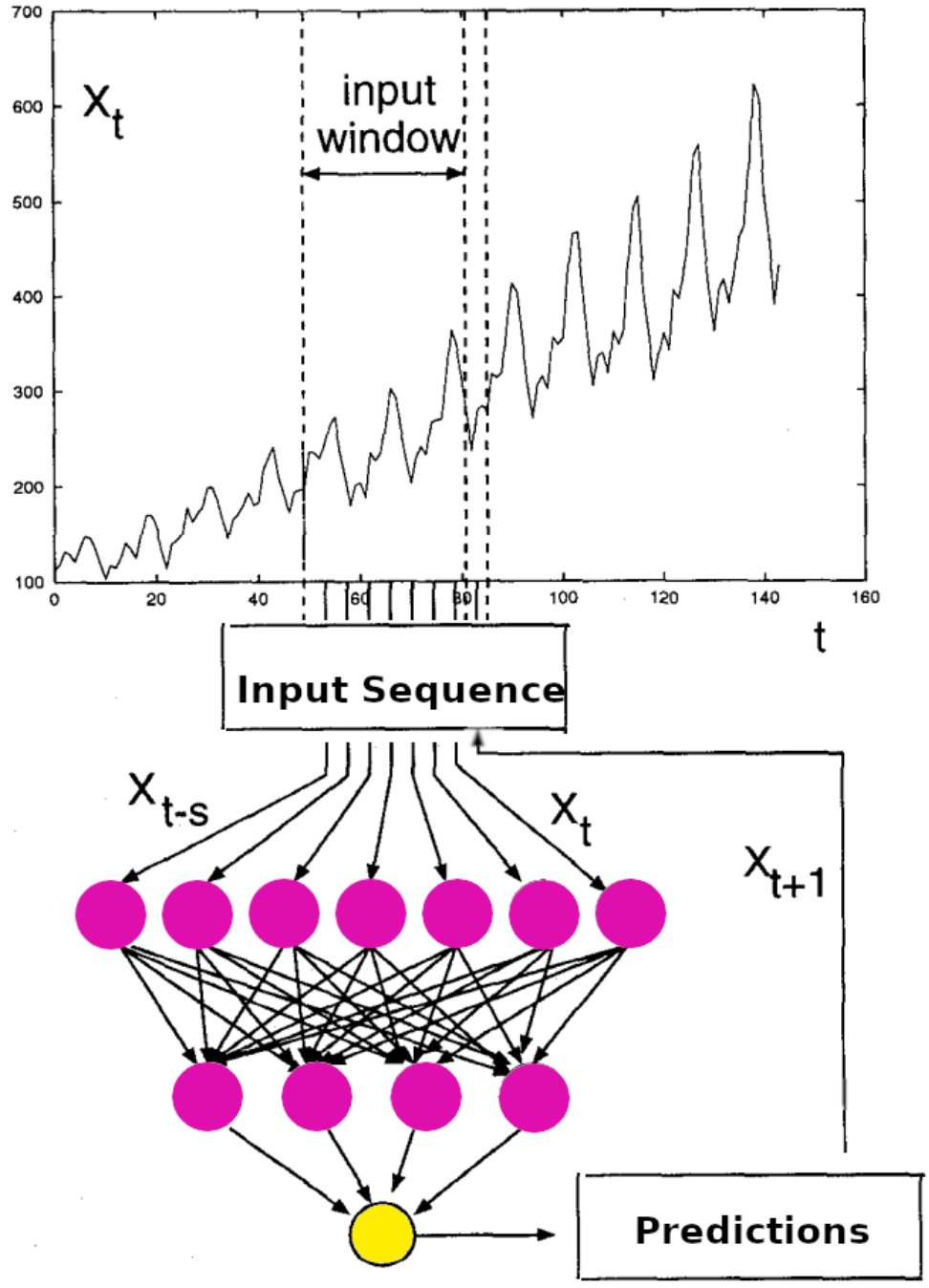

Figure 9.3: A sequence of a time series of stock prices being fed into an ANN in order to predict its future value. This multilayer perceptron does not exploit the principle of locality to the same degree as a RNN (LSTM), but the figure shows the essence of using ANNs to predict time series data. The input layer propagates the input values to the hidden layers and then to the output layer. The single output value represents the predicted next value of the time series, x_{t+1}.

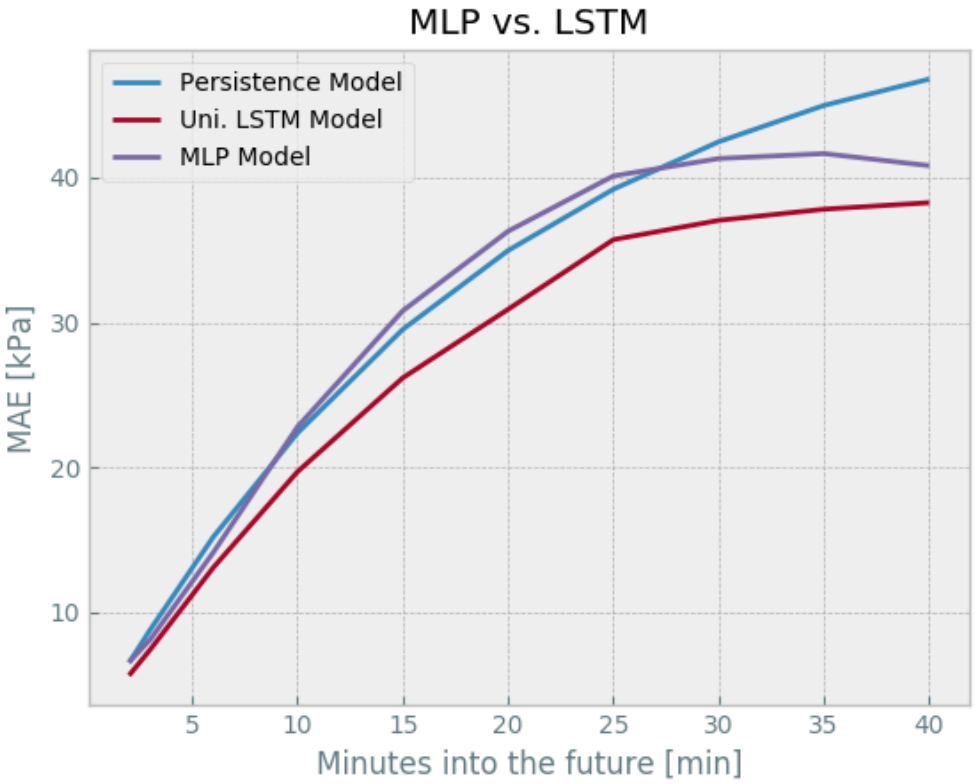

Figure 9.4: The MLPs performs consistently worse than the LSTM and is even outperformed by the persistence model when predicting between 10 to 25 minutes into the future of PDT2002's values.

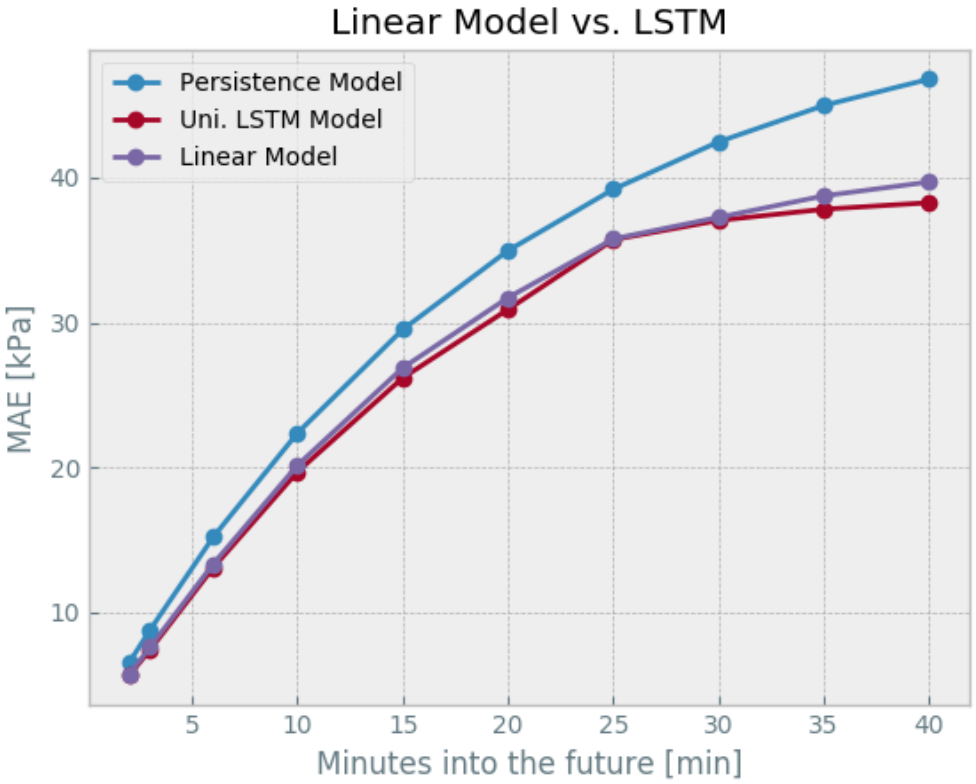

Figure 9.5: The figure shows the average prediction error of the univariate LSTM model from figure 9.4 and a Linear Prediction model from table 9.3. Although the LSTM is dominating with its smaller prediction errors, it is not by much.

Chapter 10

Multivariate Analysis Methods and Results

For our multivariate analysis we trained LSTM based neural networks. During the analysis we experimented with how many variables to use as inputs as well as adding regularization and embeddings to our networks. We found that the covariance achieved better results than mutual information when choosing the most correlated input features available. We also found that adding more than ten feature variables to our LSTM at the same time was ineffective in reducing the prediction error. The multivariate LSTMs predict the near future much more accurately than our univariate models, while accurate long term predictions were harder to improve.

10.1 Feature Selection Results

In our predictions we assumed that the same hyperparameter values that were optimal for univariate LSTMs also benefit the multivariate ones. We therefore set the timestep size to 60 seconds and the input sample size of each input feature to 20 timesteps.

Our first multivariate models were trained as an attempt to find out whether to use covariance or mutual information filters to decide which input features we want. In section 7.3 we talked about correlation filters like covariance and mutual information and how we can use them to measure correlation between features. We calculated the covariance and MI correlation between PDT2002 and all other features in the same area. Subsequently, we created two models with ten input features each. One model was fed the ten input features with the highest covariance with PDT2002, and the other was fed ten features with the highest mutual information.

In table 10.1 one can see how the models fared against each other. It's evident that covariance is the better feature selector.

Table 10.1: The table shows the differences in prediction error w.r.t. correlation type used to choose inputs. The two models each possess ten different input features. These features were chosen by either looking at their Pearson or MI correlation w.r.t. PDT2002. The results show an overall improvement when using features with high Pearson correlation as inputs instead of mutual information. Both models perform better than the univariate LSTM on short term predictions, but are slightly worse at long term predictions.

Future[min]	**MAE, N = 10, MI**	**MAE, N = 10, P**
2	4.33	**4.14**
3	6.10	**6.01**
6	11.90	**11.47**
10	18.20	**18.03**
15	25.50	**25.18**
20	30.37	**30.02**
30	37.06	**35.84**
35	38.05	**37.07**
40	38.44	**38.31**

10.2 Number of Features

The most defining aspect of our multivariable models is that they use other features than PDT2002 (as well as PDT2002) in order to predict PDT2002's future. How many of these features to put in can be approximated with a little testing. In the last section, in table 10.1, we already created benchmark results for a model with ten input features. To get a better understanding of how many features we should use as inputs, we created three more models, one with five input features, one with 15 and one with 20. The features are chosen in accordance with their Pearson correlation score w.r.t. PDT2002. In table 10.2 we have listed which number of input features were most ideal for predicting a specific distance into the future. The number of input features which consistently leads to the smallest error is 10, which is why we chose to go with this amount in our later models.

Number of Input Features Responsible for Smallest Prediction Errors

Future[min]	**Features**	**MAE [kPa]**
2	10	4.14
3	10	6.01
6	15	11.44
10	15	17.99
15	10	25.18
20	10	30.02
25	5	34.29
30	10	35.84
35	10	37.07
40	15	38.25

Table 10.2: The table shows us which model came out on top when trying to predict the value of PDT2002 n minutes into the future. Most often the model with ten input features has the smallest error, while the model with 15 input features dominates half as often.

10.3 Regularization - Dropout

In section 5.5.2 we briefly explained the simple, but effective regularization method called Dropout. We notice little to no change with traditional regularization methods such as LASSO/L2 regularization. Dropout, however, gave notable improvements to our multivariate LSTMs. Table 10.3 compares two models, one with and one without Dropout between the LSTM layer and the feed forward layer. The probability for a Dropout event to occur for a node during a forward/backpropagation is set to 25%. Both models are given the same ten input features chosen with our covariance feature selection scheme. The results are mixed, some predictions have improved and some have not, but it would seem Dropout performs slightly better on average.

10.4 Stacked LSTMs

Just like MLPs consist of feed forward layers stacked after one another, so can LSTM layers be stacked. By LSTM layer, we mean the series of LSTM cells which make up a basic LSTM network. In an LSTM layer, each cell represents a timestep from our input sample and is fed all feature data belonging to that timestep. In a basic LSTM layer, the inputs are propagated forward to the subsequent cell. In a stacked LSTM, there are LSTM cells both in the subsequent horisontal position and vertical position. Our stacked LSTM model looks quite similar to the one in figure 10.1,

Table 10.3: Table shows the prediction errors of the same LSTM network, with and without Dropout between the LSTM layer and the output layer. Most predictions have improved when adding Dropout to the model. The predictions happening 20, 30 and 35 minutes into the future have, however, worsened. In spite of this, we feel like Dropout is the way to go forward.

Future[min]	LSTM, MAE [kPa]	LSTM+D, MAE [kPa]
2	4.14	**3.91**
3	6.01	**5.77**
6	11.47	**11.17**
10	18.03	18.04
15	25.18	**24.79**
20	**30.02**	30.48
25	34.72	**34.16**
30	**35.84**	36.43
35	**37.07**	37.28
40	38.31	**38.12**

except that ours has a feed forward layer connected to the LSTM output.

We notice an increase in prediction error when using a stacked LSTM with three LSTM layers or more, but using two layers gave decent results. The results were however not an improvement overall and so we assume the extra computational cost that comes with the extra LSTM cells are not worth the small potential improvement. In table 10.4 we present a side by side comparison of our LSTM model with Dropout and our stacked LSTM model, which also uses Dropout after each LSTM layer.

Table 10.4: The Stacked LSTM seems to perform just as well as the simple LSTM. Of the forecasts, the stacked LSTM outperforms only half of the benchmarks from the simple multivariate LSTM. Both models use Dropout.

Future[min]	Stacked LSTM, MAE [kPa]	LSTM+D, MAE [kPa]
2	4.04	**3.91**
3	**5.67**	5.77
6	11.27	**11.17**
10	**17.94**	18.04
15	24.89	**24.79**
20	**29.72**	30.48
25	34.71	**34.16**
30	**36.26**	36.43
35	37.43	**37.28**
40	**38.08**	38.12

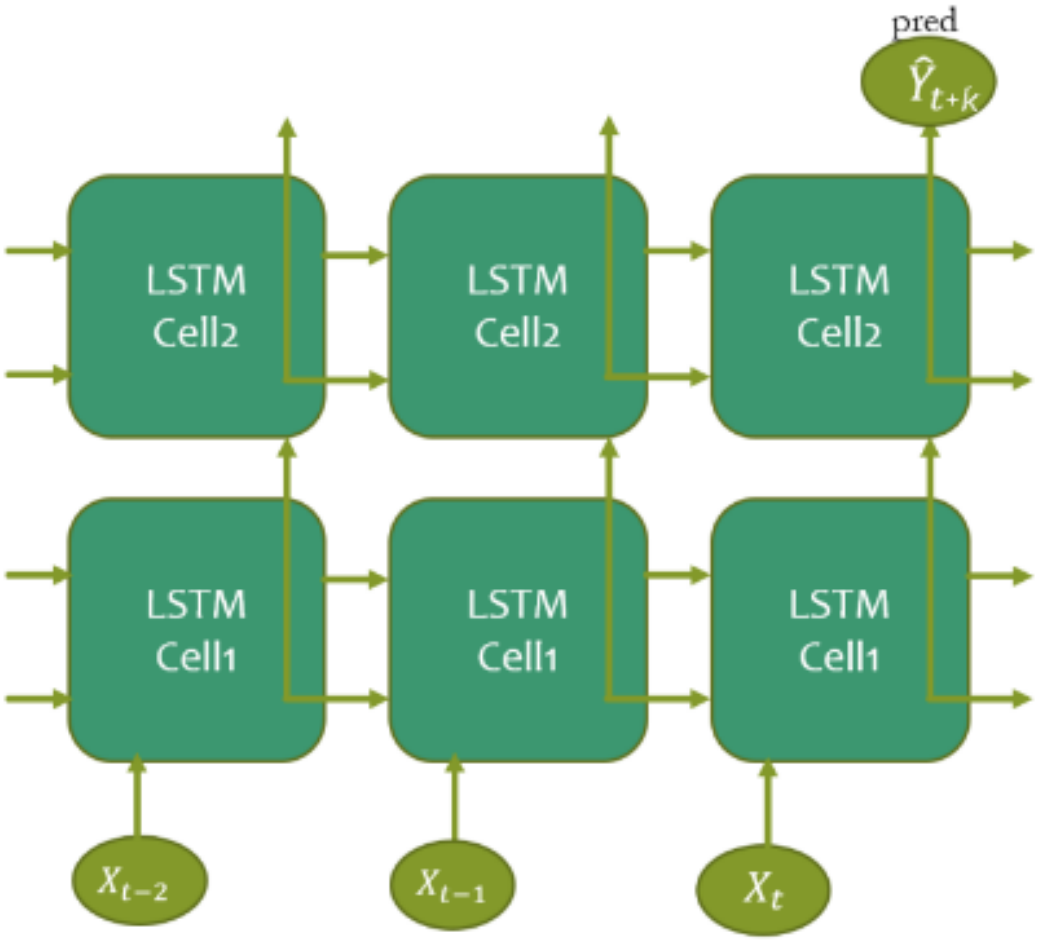

Figure 10.1: Two LSTM layers stacked on top of each other, forming a Stacked LSTM. In our model, the output of the top right cell is propagated through a feed forward layer whose output is evaluated as the network's prediction. The image is based on a figure found in Tong et al. (2018).

10.5 Embeddings and Time Data

In this section we introduce both new input features and additional architecture to our models in an attempt to improve our predictions. By architecture, we mean changes to the neural networks instead of the features. The new input features we began introducing were categorical features. To input this data, we created an embedding which transforms a sparse encoded vector into a set of 64 hidden nodes which we concatenated with the output of the LSTM. A feed forward layer connected this merged layer to the output. We included Dropout between the merged layer and the feed forward layer.

Most of the categorical data we made were one hot encoded vectors which represent the timestamp of the most recent input timestep, t (the timestep corresponding to the rightmost input of the LSTM in figure 10.1). We assume that given awareness of the time of the prediction, the network might pick up new patterns in the data that are unique to a certain timeperiod of the year/week/day. By this logic, we also made a one hot vector of length one, which describes whether or not today is a national holiday or a national sports event.

10.5.1 Categorical Time Data

The motivation behind improving our models with categorical data came from our hearing about CatBoost (Dorogush et al. 2017), a powerful machine learning method that can be used for both regression and classification using categorical data. Instead of using methods like CatBoost and other ”decision tree“ methods in our thesis, we have stuck with neural networks to keep things manageable. For neural networks, embeddings and encoding methods are required to make categorical data into something numerical and by thereby applicable.

We feed our categorical data into our network by encoding it into sparse vectors which we then send through an embedding. As we explained in section 5.4, an embedding is a weight matrix which maps a sparse binary vector to a real valued vector. The sparse vectors we input alongside our features are concatenated one hot vectors. Each one hot vector is a representation of e.g. which month it is or what time it was when the feature's value was measured. Say the input sample we are putting into our model was measured April 4th, 2018 at 12:31 PM on a Wednesday. We can then, parallell to feeding our LSTM the numerical measurement, feed our embedding a sparse vector corresponding to the date and time that the numerical measurement was observed.

We observed in our results that models fed different combinations of time data inputs had both positive and negative effects on the network's ability to predict

PDT2002's values. With inputs such as a one hot vector representing the measurements month, time of day, weekday etc. we found that a combination of these created a general improvement in our results. This optimal combination was a sparse input vector/feature representing both the: day in the month, weekday in the week, month in the year and minute of the hour which corresponded to when the numerical input features of the input sample were measured. In table 10.5 is a side by side comparison of the new LSTM model and the basic LSTM with Dropout. In the table we observed improvements in both long term and short term predictions, however, most prediction errors between 6 and 35 minutes have worsened.

Table 10.5: The LSTM model with Dropout is here given an embedding. Most errors have increased with the introduction of the embedding, but the error of predictions of PDT2002, 40 minutes into the future, has significantly decreased. For this reason, we feel like LSTMs with an embedding has the potential to become good long term prediction models.

Future[min]	**Embedding + LSTM, MAE [kPa]**	**LSTM+D, MAE [kPa]**
2	3.90	3.91
3	**5.60**	5.77
6	11.32	**11.17**
10	18.44	**18.04**
15	25.47	**24.79**
20	30.86	**30.48**
25	34.86	**34.16**
30	**36.36**	36.43
35	37.67	**37.28**
40	**37.77**	38.12

10.5.2 Special Features

A big decision and assumption we made in this thesis was to use features close to tag PDT2002 to predict its future measurements because we assumed tags who are close are correlated. We have not used features from other places in Oslo. A set of features was given to us, by the utility company responsible for the tags, representing tags associated with adjacent heating factories. These do not share area code with PDT2002, but were thought to affect it nonetheless. We ran a Pearson correlation filter over each of the features w.r.t. PDT2002 and fed our LSTM+embedding model an increasing amount of the most correlated tags to see if they had a positive effect. In addition to these tags, we also still used the ten tags from PDT2002s district as inputs.

10.5.3 LSTM with Embedding and Special Features

We saw improvements in long term predictions when adding four of the special features most correlated to PDT2002 into our models. Fewer or more than four special features had negative effects on the overall predictions (1 to 6 special features were tested). In table 10.6 is a comparison between the new model trained on the four special features vs. the older LSTM from section 10.5. Both models use embeddings. The new model is fed information about which month, weekday and day of the month took place, as well as holiday information. The older model was also informed which minute it was, during the hour of the measurement, something we excluded from the newer model to enhance predictions.

Table 10.6: The left column lists the mean prediction errors of the previous Embedding LSTM we presented the results of. The right column lists the errors of the same model, except that the LSTM cells also are given features adjacent to surrounding heat factories which we have called ”special features“.

Future[min]	Embedding, MAE [kPa]	Embedding + Special, MAE [kPa]
2	3.90	**3.84**
3	**5.60**	5.89
6	11.32	11.34
10	18.44	**18.03**
15	25.47	**24.87**
20	30.86	**30.26**
25	34.86	**33.82**
30	**36.36**	36.49
35	37.67	**37.05**
40	37.77	**37.55**

In a last attempt to improve our predictions, we are going to take our short term prediction models and train them on differenced data, which we earlier observed to decrease the prediction error of our 2 and 3 minute predictions in table 9.1. We managed to reproduce the same effect with the current LSTM we have been feeding special features. In table 10.7 we have put the prediction errors of the model before and after differencing the data and it's clear that differencing has a positive effect on short term predictions.

10.5.4 Basic LSTM with Dropout and Special Features

Not only did the special features improve the LSTMs with embeddings, they also improved the LSTMs without it. Below, in table 10.8, is a side by side comparison of the LSTM with Dropout from table 10.3 next to the same model being fed the

Table 10.7: Differencing our features give us better prediction errors when predicting into the near future (2-3 minutes), but not from 6 minutes onward. The model is the same multivariate LSTM from table 10.6, trained on special features.

Future[min]	Differenced, MAE [kPa]	Non-Differenced, MAE [kPa]
2	**3.52**	3.84
3	**5.35**	5.89

4 special features we used in the previous section. While these results are overall inferior to the LSTM + embedding model, they still tell us something about the special features. The results confirm for a second time that the features from the surrounding heat factories have the ability to enhance our long term predictions.

Table 10.8: This table describes the effect of special features on univariate LSTMs instead of multivariate ones. Feeding special features into a univariate LSTM model without embeddings seems to improveme the prediction errors.

Future[min]	LSTM+D+Special, MAE [kPa]	LSTM+D, MAE [kPa]
2	3.94	**3.91**
3	**5.67**	5.77
6	11.21	**11.17**
10	18.18	**18.04**
15	24.94	**24.79**
20	**30.27**	30.48
25	34.34	**34.16**
30	36.44	36.43
35	37.68	**37.28**
40	**37.88**	38.12

10.6 Final Comparisons

To properly showcase our results we will in this section benchmark all the predictions our models have made against each other. As evaluation metrics we will use the mean absolute test error, which is the metric we have been using to present the prediction errors with up until now. We will also show the absolute median error. The mean errors are presented in table 10.9 and the median errors in table 10.10. Overall the multivariate LSTMs have the best prediction errors. The LSTM with an embedding tends to have a smaller median error, and a very similar mean absolute error, compared to the LSTM without an embedding.

Table 10.9: The table shows the MAE of our univariate models (MLP and Linear Prediction) and our mulitvariate models(LSTM, LSTM+E). The LSTMs both contain Dropout and have been trained on special features. LSTM+E is shorthand for LSTM with Embedding and the "Baseline" is the persistence model. We have boldfaced the best predictions.

Future[min]	MLP	Linear	LSTM	LSTM+E	Baseline
2	6.55	5.69	3.94	**3.84**	6.50
3	8.09	7.58	**5.67**	5.89	8.77
6	14.11	13.32	**11.21**	11.34	15.19
10	22.79	20.14	18.18	**18.04**	22.37
15	30.80	26.86	24.94	**24.79**	29.51
20	36.29	31.71	**30.27**	30.48	34.96
25	40.10	35.76	34.34	**33.82**	39.19
30	41.31	37.25	**36.44**	36.49	42.46
35	41.65	38.73	37.68	**37.05**	44.97
40	40.81	39.69	37.88	**37.55**	46.79

From table 10.9 and 10.10 we see the model that makes the best predictions for most of the distances forward in time is the multivariate LSTM equipped with an embedding (LSTM+E). In both tables we can that the LSTM without the embedding performs best on predictions 2, 3, and 30 minutes forward in time, but this is not enough to really compete with LSTM+E. Because we like graphs we two more (figure 10.2 and 10.3), with all models except MLP, visualizing both the mean and median prediction errors.

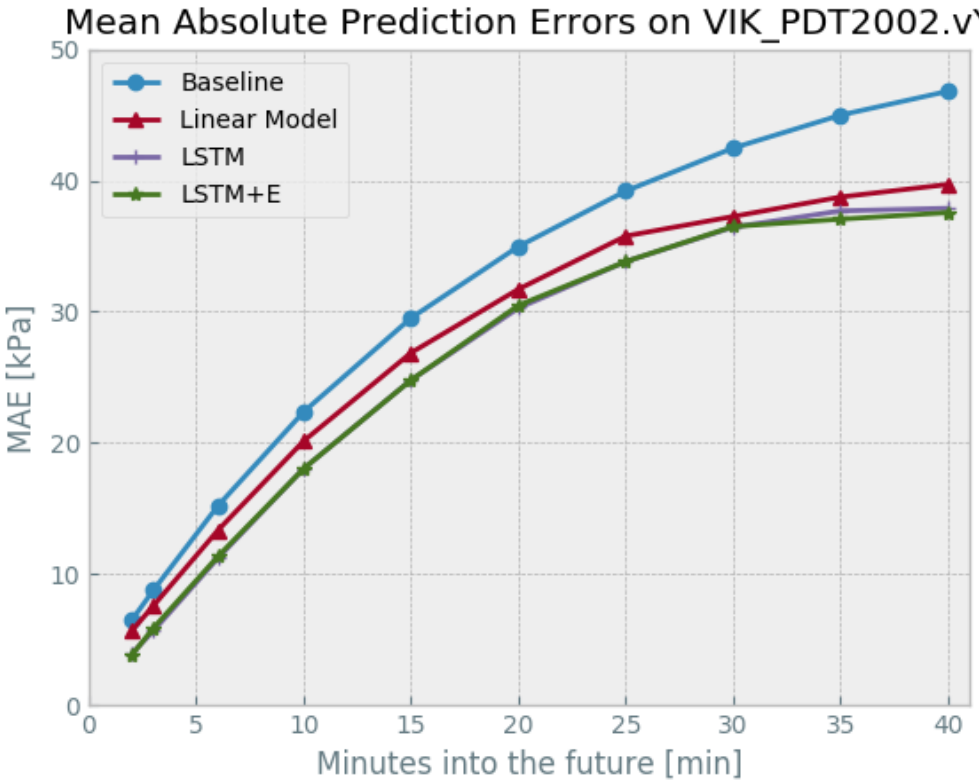

Figure 10.2: Comparison of our models' mean squared prediction error. The multivariate LSTMs dominate the linear model, but it is hard to visually distinguish the performance of the LSTMs themselves.

Table 10.10: The table shows the Median Absolute Error of our models. The LSTMs listed here are multivariate.

Future[min]	MLP	Linear	LSTM	LSTM+E	Baseline
2	5.16	4.04	2.64	**2.57**	4.36
3	5.38	5.19	**3.85**	3.92	5.67
6	10.09	8.83	**7.29**	7.36	9.42
10	16.38	13.20	11.75	**11.55**	13.46
15	21.13	17.35	15.93	**15.76**	17.46
20	24.39	20.31	19.17	**19.10**	20.49
25	25.84	22.23	21.35	**21.03**	22.85
30	26.95	23.69	**22.74**	23.01	25.10
35	27.49	24.68	23.50	**23.31**	26.54
40	26.43	25.36	23.43	**23.23**	27.78

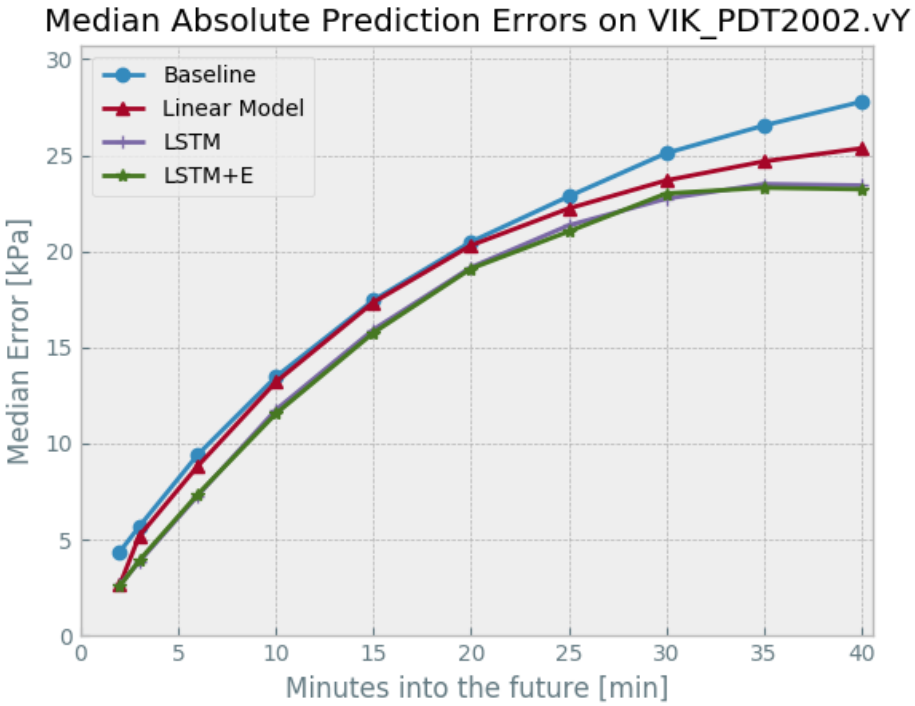

Figure 10.3: Comparison of our models' median absolute prediction error.

Part IV

Summary

Chapter 11

Discussion and Conclusion

In this chapter we discuss our methods and results to see if we can draw conclusions from them. We begin by discussing the quality of our methods and analysis, followed by a closer examination of the results. Before we draw any conclusions we will also discuss related works in the litterature on the subject of using ANNs for time series forecasting to see if our results are in agreement with similar experiments.

11.1 Discussion

While our analysis has resulted in seemingly competent models, some of the steps along the way deserve some scrutiny. First of all, we interpolated our data measurements to the closest timestep (minute) they belonged to, something that may have removed important information from the data. As we mentioned in the data pipeline chapter (6), the data actually has a frequency of 1 Hz. Resampling the data to be 1/60 Hz, or less, is likely going to remove some patterns from the data, but working with data on a higher resolution was not feasible. This is mainly due to our limited hardware capabilities.

Despite our interpolation potentially removing information from the data, our models still found patterns in them. The models we made outperformed the persistence model, meaning they did manage to learn from the processed data we fed them. Downsampling lead to holes in our data, but these were filled with forward filling. This way of filling holes in sequences creates data that is coherent with reality. This is because of the event-driven nature of the sensors, meaning the gaps are there due to no noticeable change having taken place since the previous data measurement.

In our analyses, we have assumed that the hyperparameters that worked well for

univariate LSTMs also were good for multivariate LSTMs. Even though thr results show that multivariate LSTMs perform better than the univariate ones, a new grid search of hyperparameter values might have improved thr results even more. In the multivariate analysis we also introduced new hyperparameters, e.g. number of local input features, Dropout, number of categorical input features and number of special features. We did not run a large grid search over all these parameters, instead we optimized w.r.t. one hyperparameter at a time, which might have had negative consequences for models. In hindsight, we could have tried to make a random hyperparameter search.

All model permutations[1] were trained once to predict a specific distance into the future, creating a notable potential for uncertainty in our results. Doing multiple runs for each model to analyze the variance in the prediction error for each one was too time consuming. This is because of the high number and variations of hyperparameters we had to test for, resulting in thousands of models needing to be trained. Although there is reason for worry, the models are each trained ten different times to predict ten different distances into the future. This means that if a model has a lower prediction error that another one, for all ten prediction distances into the future, the winning model has proved itself ten times, making its performance anything else than a coincidence.

The results show that LSTM models performed consistently better than the linear autoregressive models. At every prediction interval the multivariate LSTMs had a lower prediction error than the univariate LSTMs, which in turn gave more accurate predictions than the linear models. Multiple papers in the forecasting litterature have compared the predictions of linear autoregressive models to ANNs, often MLPs, and the results often favor ANNs, like Balkin and Ord (2000), or Jain and Kumar (2007). What these papers have in common is that they find that traditional forecasting models are linear in nature and thus struggle with non-linear data.

While LSTMs performed better than linear AR models, MLPs performed worse. The reason behind this might be that we did not perform a proper optimization of the MLPs architecture, by simply going with 128 hidden nodes and one hidden layer from the start. Optimizing a neural network model is more strenous than traditional time series forecasting models. Perhaps the MLP would have performed better if we had put more resources into optimizing the structure of it. There are cases where MLPs perform worse than linear models, either when modeling linear time series or when simulating physical phenomena (Fishwick 1989). Linear time series, however, are rare to come by in real world data.

The success of models that run on categorical data, such as Catboost (Dorogush

[1]By permutation we mean a model with a unique set of hyperparameters and type (MLP/AR/LSTM).

et al. 2017), still has us convinced that we can improve our models through the use of more categorical data. However, categorical variables provided only some minor improvements to the predictions of the LSTM model. We firmly believe that there is much more potential to be gained from using categorical input data than was achieved in our limited investigations. The physical system we have been making predictions for is highly dependent on human activity from both customers and those working on the system. In order to foresee effects on the system made by these agents, we need to feed the network categorical information related to human activity, such as time and whether or not people are working that day.

11.2 Conclusion

In this thesis, we have created a data pipeline for industrial data, implemented various neural network models for forecasting and compared their prediction accuracy with traditional linear AR models. The short-term forecast was performed on an industrial dataset from a DHS. There was also some experimentation with categorical data as inputs for the neural network models. Among the proposed neural network models, the LSTM models provide a promising alternative to classical forecasting methods, like linear AR models. From the results, we can saw that the LSTM models give smaller prediction errors than linear AR models. As demonstrated in the thesis, LSTMs require multiple correlated input features in order to achieve a minimal prediction error. Feeding categorical features as inputs with an embedding brought a minor, but not significant, improvement to the LSTM's prediction error.

11.3 Perspective

Our results do not prove the superiority of all ANNs over traditional forecasting models, but they strengthen the hypothesis: Neural networks, such as LSTMs, are better-suited models for time series prediction than their linear natured ancestors (e.g. linear regression). LSTMs' predictions in particular have been shown to outperform complex AR models, known as ARIMA (Box and Jenkins 1970), as well as RNNs and SVMs (Cortes and Vapnik 1995) in Zhao et al. (2017). However, ML ensemble methods like xgboost (Chen and Guestrin 2016), LightGBM (Ke et al. 2017) and CatBoost, are consistently winning both classification and forecasting competitions at the moment (Rajkumar 2019). If we were to do further analysis on the subject of time series forecasting, we would compare neural networks with modern ensemble methods.

Appendices

Appendix A

OLS Derivation

If you remember univariate optimization from calculus, you know that we can set a function's derivatives equal to zero to find which parameter values return the function's minima. The point of this appendix is find an analytical expression for any OLS model. Initially, we take the gradients of the OLS cost function w.r.t. its parameters, β and β_0, from eq. (3.4) and 3.5 and set their r.h.s. to zero

$$\frac{\partial S_{ssr}}{\partial \beta}(\beta, \beta_0) = -2\sum_{i=1}^{n}\left(t_i T(t_i) - \beta_0 t_i - \beta t_i^2\right) = 0, \tag{A.1}$$

$$\frac{\partial S_{ssr}}{\partial \beta_0}(\beta, \beta_0) = -2\sum_{i=1}^{n}\left(T(t_i) - \beta_0 - \beta t_i\right) = 0. \tag{A.2}$$

Starting with (A.2), we can remove the -2 coefficient and use the fact that $\sum_i^n x = n\bar{x}$ to get the following solution for β_0

$$\begin{aligned}
\frac{\partial S_{ssr}}{\partial \beta_0}(\beta, \beta_0) = 0 \implies \\
-2\sum_{i=1}^{n}\left(T(t_i) - \beta_0 - \beta t_i\right) = 0 \implies \\
n\beta_0 + n\beta\bar{t} - n\bar{T} = 0 \implies \\
\beta_0 = \bar{T} - \beta\bar{t}.
\end{aligned} \tag{A.3}$$

The optimal value for the slope variable β that minimizes our error can now be found by inserting the expression for β_0 from (A.3) into (A.1)

$$\begin{aligned}
\frac{\partial S_{ssr}}{\partial \beta}(\beta, \beta_0) = 0 \implies \\
-2\sum_{i=1}^{n}\left(t_i T(t_i) - (\bar{T} - \beta\bar{t})t_i - \beta t_i^2\right) = 0 \implies \\
\sum_{i=1}^{n}(t_i T(t_i)) - \sum_{i=1}^{n}\left(\bar{T}t_i\right) + \sum_{i=1}^{n}\left(\beta\bar{t}t_i\right) - \sum_{i=1}^{n}\left(\beta t_i^2\right) = 0 \implies \\
\sum_{i=1}^{n}(t_i T(t_i)) - n\bar{T}\bar{t} + n\beta\bar{t}^2 - \beta\sum_{i=1}^{n}\left(t_i^2\right) = 0 \implies \\
\sum_{i=1}^{n}(t_i T(t_i)) - n\bar{T}\bar{t} = \beta\left[\sum_{i=1}^{n}\left(t_i^2\right) - n\bar{t}^2\right].
\end{aligned}$$

Solving this equation for β yields us a steadily prettier expression

$$\beta = \frac{\sum_{i=1}^{n}(t_i T(t_i)) - n\bar{T}\bar{t}}{\sum_{i=1}^{n}(t_i^2) - n\bar{t}^2}$$

$$\begin{aligned}
\beta &= \frac{\sum_{i=1}^{n}(t_i T(t_i)) - n\bar{T}\bar{t}}{\sum_{i=1}^{n}(t_i^2) - n\bar{t}^2} \\
&\quad \text{(see section A.1)} \\
&= \frac{\sum_{i=1}^{n}(t_i - \bar{t})(T(t_i) - \bar{T})}{\sum_{i=1}^{n}(t_i - \bar{t})^2} \\
&= \frac{\mathrm{Cov}(t, T(t))}{\mathrm{Var(t)}}. \qquad \text{(A.4)}
\end{aligned}$$

With (A.4) we can now write a closed form solution for β_0 too

$$\beta_0 = \bar{T} - \bar{t}\frac{\mathrm{Cov}(t, T(t))}{\mathrm{Var(t)}}. \qquad \text{(A.5)}$$

A.1 Side Proof

The point of this subsection is to derive eq. (A.4), or rather prove the following equivalence

$$\frac{\sum_{i=1}^{n}(t_i T(t_i)) - n\bar{T}\bar{t}}{\sum_{i=1}^{n}(t_i^2) - n\bar{t}^2} = \frac{\sum_{i=1}^{n}(t_i - \bar{t})(T(t_i) - \bar{T})}{\sum_{i=1}^{n}(t_i - \bar{t})^2}. \qquad \text{(A.6)}$$

We can prove that the l.h.s. is equal to the r.h.s. in eq. (A.6) by individually proving that its numerators are equal and that its denominators are equal. The new problem now becomes to prove eq. (A.7) and (A.8). Notice that we have changed notation from t to x and T to y.

$$\sum_{i=1}^{n}(x_i^2) - n\bar{x}^2 = \sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{A.7}$$

$$\sum_{i=1}^{n}(x_i y_i) - n\bar{x}\bar{y} = \sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \tag{A.8}$$

We prove both equations by deriving the l.h.s. from the r.h.s., starting with A.7. We begin the derivation by expanding A.7s r.h.s. and the use the fact that $n\bar{x} = \sum_i x_i$

$$\begin{aligned}
\sum_{i=1}^{n}(x_i - \bar{x})^2 &= \sum_{i=1}^{n}(x_i^2 - 2x_i\bar{x} + \bar{x}^2) \\
&= \sum_{i=1}^{n}(x_i^2) - 2\bar{x}\sum_{i=1}^{n}(x_i) + n\bar{x}^2 \\
&= \sum_{i=1}^{n}(x_i^2) - 2n\bar{x}^2 + n\bar{x}^2 \\
&= \underline{\underline{\sum_{i=1}^{n}(x_i^2) - n\bar{x}^2}}.
\end{aligned} \tag{A.9}$$

As you can see, A.7's right hand side is equivalent with its left hand side.

On to proving A.8. Here, we also exploit the fact that $\sum_i x = n\bar{x}$. We start with the right hand side of A.8 to derive its left hand side

$$\begin{aligned}
\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) &= \sum_{i=1}^{n}(x_i y_i - x_i\bar{y} - y_i\bar{x} + \bar{x}\bar{y}) \\
&= \sum_{i=1}^{n}(x_i y_i) - \sum_{i=1}^{n}(x_i\bar{y}) - \sum_{i=1}^{n}(y_i\bar{x}) + \sum_{i=1}^{n}(\bar{x}\bar{y}) \\
&= \sum_{i=1}^{n}(x_i y_i) - \bar{y}\sum_{i=1}^{n}(x_i) - \bar{x}\sum_{i=1}^{n}(y_i) + n\bar{x}\bar{y} \\
&= \sum_{i=1}^{n}(x_i y_i) - n\bar{x}\bar{y} - n\bar{x}\bar{y} + n\bar{x}\bar{y} \\
&= \underline{\underline{\sum_{i=1}^{n}(x_i y_i) - n\bar{x}\bar{y}}}.
\end{aligned} \tag{A.10}$$

Bibliography

ABB (2013). *Big Data and decision-making in industrial plants*. URL: https://new.abb.com/cpm/production-optimization/big-data-analytics-decision-making.

Apergis, Nicholas and James E. Payne (Jan. 2010). "Renewable energy consumption and economic growth: Evidence from a panel of OECD countries". In: *Energy Policy* 38.1, pp. 656–660. DOI: 10.1016/j.enpol.2009.09.002. URL: http://dx.doi.org/10.1016/j.enpol.2009.09.002.

Balkin, Sandy D. and J.Keith Ord (Oct. 2000). "Automatic neural network modeling for univariate time series". In: *International Journal of Forecasting* 16.4, pp. 509–515. DOI: 10.1016/s0169-2070(00)00072-8. URL: http://dx.doi.org/10.1016/S0169-2070(00)00072-8.

Box, George E. P. and M. Jenkins (1970). *Time Series Analysis: Forecasting and Control*. DOI: 10.2307/2284112.

"Stationary Processes" (2002). In: *Introduction to Time Series and Forecasting*. Ed. by Peter J. Brockwell and Richard A. Davis. New York, NY: Springer New York, pp. 45–82. ISBN: 978-0-387-21657-7. DOI: 10.1007/0-387-21657-X_2. URL: https://doi.org/10.1007/0-387-21657-X_2.

Bryson, Arthur E. (1961). "A gradient method for optimizing multi-stage allocation processes." In: *InProc.Harvard Univ. Symposium on digital computers and their applications*. The English translation by Ada Lovelace from Scientific Memoirs that was consulted can be found here: http://www.fourmilab.ch/babbage/sketch.html.

Chen, Tianqi and Carlos Guestrin (2016). "XGBoost: A Scalable Tree Boosting System". In: *CoRR* abs/1603.02754. arXiv: 1603.02754. URL: http://arxiv.org/abs/1603.02754.

Cortes, Corinna and Vladimir Vapnik (Sept. 1995). "Support-vector networks". In: *Machine Learning* 20.3, pp. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018. URL: https://doi.org/10.1007/BF00994018.

Dorogush, Anna Veronika et al. (2017). "Fighting biases with dynamic boosting". In: *CoRR* abs/1706.09516. arXiv: 1706.09516. URL: http://arxiv.org/abs/1706.09516.

Edell, Laura (2015). *Is Machine Learning the New EPM Black?* URL: https://scorecardstreet.wordpress.com/2015/12/09/is-machine-learning-the-new-epm-black/.

EU (2013). *European Technology Platform on Renewable Heating and Cooling Strategic research and innovation agenda for renewable heating and cooling*. EU.

Fahlman, S. E. (Sept. 1988). "An Empirical Study of Learning Speed in Backpropagation Networks". In: *Technical Report CMU-CS-88-162*.

Fishwick, P. A. (1989). "Neural Network Models in Simulation: A Comparison with Traditional Modeling Approaches". In: *Proceedings of the 21st Conference on Winter Simulation*. WSC '89. Washington, D.C., USA: ACM, pp. 702–709. ISBN: 0-911801-58-8. DOI: 10.1145/76738.76828. URL: http://doi.acm.org/10.1145/76738.76828.

Glorot, Xavier and Yoshua Bengio (May 2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

Google (2019). *Embeddings*. URL: https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture.

Granger, C. W. J. and Roselyne Joyeux (1980). "An introduction to long-memory time series models and fractional differencing". In: *J. Time Ser. Anal.* 1.1, pp. 15–29. ISSN: 0143-9782. DOI: 10.1111/j.1467-9892.1980.tb00297.x. URL: https://doi.org/10.1111/j.1467-9892.1980.tb00297.x.

Graves, Alex and Kelly Clancy (Apr. 2019). "Unsupervised learning: the curious pupil". In: URL: https://deepmind.com/blog/unsupervised-learning/.

Guyon, Isabelle et al. (Jan. 2002). "Gene Selection for Cancer Classification using Support Vector Machines". In: *Machine Learning* 46.1, pp. 389–422. ISSN: 1573-0565. DOI: 10.1023/A:1012487302797. URL: https://doi.org/10.1023/A:1012487302797.

Han, Jun and Claudio Moraga (1995). "The influence of the sigmoid function parameters on the speed of backpropagation learning". In: *From Natural to Artificial Neural Computation*. Ed. by José Mira and Francisco Sandoval. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 195–201. ISBN: 978-3-540-49288-7.

Herbert Robbins, Sutton Monro (1951). "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* 22(3). http://www.jstor.org/stable/2236626, pp. 400–407.

Hjorth-Jensen, Morten (2018). *Regression*. URL: https://compphysics.github.io/MachineLearningMSU/doc/pub/Regression/html/Regression.html.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735. eprint: https://doi.org/10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White (Jan. 1989). "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5, pp. 359–366. DOI: 10.1016/0893-6080(89)90020-8. URL: http://dx.doi.org/10.1016/0893-6080(89)90020-8.

Hurst, H. E. (1951). "Long-Term Storage Capacity of Reservoirs". In: *Trans. Amer. Soc. Civil Eng.* 116, pp. 770–799. URL: https://ci.nii.ac.jp/naid/10014568424/en/.

IPCC (2007). *Climate Change 2007: Synthesis Report. Contribution of Working Groups I, II and III to the Fourth Assessment Report of the Intergovernmental Panel on Climate Change*. IPCC.

ISA Codes (n.d.). https://www.engineeringtoolbox.com/isa-intrumentation-codes-d_415.html. Last Accessed: 2019-06-23.

Jain, Ashu and Avadhnam Madhav Kumar (Mar. 2007). "Hybrid neural network models for hydrologic time series forecasting". In: *Applied Soft Computing* 7.2, pp. 585–592. DOI: 10.1016/j.asoc.2006.03.002. URL: http://dx.doi.org/10.1016/j.asoc.2006.03.002.

Jiaconda (2016). *A Concise History of Neural Networks*. URL: https://new.abb.com/cpm/production-optimization/big-data-analytics-decision-making.

Johann, Georg (2010). *Sigmoids*. URL: https://commons.wikimedia.org/w/index.php?curid=11498624.

Karpathy, Andrej (2015). *The Unreasonable Effectiveness of Recurrent Neural Networks*. URL: https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Ke, Guolin et al. (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree". In: *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon et al. Curran Associates, Inc., pp. 3146–3154. URL: http://papers.nips.cc/paper/6907-lightgbm-a-highly-efficient-gradient-boosting-decision-tree.pdf.

Kelley, Henry J. (1960). "Gradient Theory of Optimal Flight Paths". In: *American Rocket Society Archive Volume 30, Number 10*.

Kingma, Diederik P. and Jimmy Ba (Dec. 22, 2014). "Adam: A Method for Stochastic Optimization". Version 9. In: arXiv: http://arxiv.org/abs/1412.6980v9 [cs.LG]. URL: http://arxiv.org/abs/1412.6980v9.

Time series forecasting using neural networks (1994). ACM Press. DOI: 10.1145/190271.190290. URL: http://dx.doi.org/10.1145/190271.190290.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (Jan. 2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25. DOI: 10.1145/3065386.

McCulloch, Warren S. and Walter Pitts (Dec. 1943). "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

Mehta, Pankaj et al. (Mar. 23, 2018). "A high-bias, low-variance introduction to Machine Learning for physicists". Version 1. In: arXiv: http://arxiv.org/abs/1803.08823v2 [physics.comp-ph, cond-mat.stat-mech, cs.LG, stat.ML]. URL: http://arxiv.org/abs/1803.08823v2.

MPE (2008). *Energy and Water Resources in Norway*. URL: https://www.regjeringen.no/globalassets/upload/oed/pdf_filer/faktaheftet/evfakta08/evfacts08_kap03_eng.pdf.

Rajkumar, Sudalai (2019). *Winning solutions of kaggle competitions*. URL: https://www.kaggle.com/sudalairajkumar/winning-solutions-of-kaggle-competitions.

Santosa, F. and W. Symes (1986). "Linear Inversion of Band-Limited Reflection Seismograms". In: *SIAM Journal on Scientific and Statistical Computing* 7.4, pp. 1307–1330. DOI: 10.1137/0907087. eprint: https://doi.org/10.1137/0907087. URL: https://doi.org/10.1137/0907087.

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Tensorflow (n.d.). *Feature Columns*. URL: https://www.tensorflow.org/guide/feature_columns.

Tong, Tao et al. (2018). "Investigating Long Short-Term Memory Neural Networks for Financial Time-Series Prediction". In: *SSRN Electronic Journal*. DOI: 10.2139/ssrn.3175336. URL: http://dx.doi.org/10.2139/ssrn.3175336.

Wyk, Andrich van (Apr. 2018). "Encoding Cyclical Features for Deep Learning". In: URL: https://www.avanwyk.com/encoding-cyclical-features-for-deep-learning/.

Yann, LeCun et al. (1998). *"Efficient backprop" in Neural networks: Tricks of the trade*. Springer.

Zhang, G. Peter (2012). "Neural Networks for Time-Series Forecasting". In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 461–477. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_14. URL: https://doi.org/10.1007/978-3-540-92910-9_14.

Zhao, Z. et al. (2017). "LSTM network: a deep learning approach for short-term traffic forecast". In: *IET Intelligent Transport Systems* 11.2, pp. 68–75. ISSN: 1751-956X. DOI: 10.1049/iet-its.2016.0208.

Zou, Kelly H., Kemal Tuncali, and Stuart G. Silverman (June 2003). "Correlation and Simple Linear Regression". In: *Radiology* 227.3, pp. 617–628. DOI: 10.1148/radiol.2273011499. URL: http://dx.doi.org/10.1148/radiol.2273011499.