



Norwegian University of
Science and Technology

A study of Machine Learning for Predictive Maintenance

A topic and programming guidance

Kåre Hartlapp Lærum

Master of Science in Mechanical Engineering

Submission date: June 2018

Supervisor: Amund Skavhaug, MTP

Norwegian University of Science and Technology
Department of Mechanical and Industrial Engineering

MASTER'S THESIS SPRING 2018

FOR

STUD.TECHN. Kåre H. Lærum

A study of Machine Learning for Predictive Maintenance

A topic and programming guidance

This Master thesis aims to provide an introduction and to be practical start guide for those interested in using Machine Learning for Predictive Maintenance. To achieve this, firstly, the topics Predictive Maintenance (PdM), Machine Learning (ML) and Transfer Learning (TL) are presented. Then, the thesis aims to provide an understanding towards ML and to give an introduction to the questions; what is ML, how does it work, and how can one build an ML model to start with initial testing?

Contact:

At the department (supervisor): Professor Amund Skavhaug, Department of Mechanical and Industrial Engineering, NTNU

Address:

NO-7491 TRONDHEIM
Norway

Org.nr. 974 767 880

Email:

mtp-info@mtp.ntnu.no

<https://www.ntnu.edu/mtp>

i. ABSTRACT

A significant potential and interest is found for Predictive Maintenance (PdM) and Machine Learning (ML). Both fields are under development and need further research. Through collaboration with the Norwegian maintenance company, Karsten Moholt, two areas of special significance for this thesis were found. Firstly, finding more information regarding the ML approach and especially the potential of Transfer Learning (TL), and secondly, understanding and building an ML model for PdM.

This thesis aims to provide an introduction and to be practical start guide for those interested in using ML for PdM. To achieve this, necessary background information has been collected, and to further learn about ML, several online courses have been completed. The online course, books and several web-guides are used to build and develop two ML models to find Remaining Useful Life (RUL) on the Turbofan Engine Degradation Data Set provided by NASA. The information and code are presented to enable the reader to understand ML and to provide a tool to start building ML models.

PdM is presented as a technique for monitoring operating condition to provide data that can ensure the maximum interval between repairs and minimize the number and cost of unscheduled machine failures. ML is considered as an important and powerful tool for finding patterns and make predictions from a vast amount of data where Neural Networks (NN) is the main technique for implementing ML. TL is considered as a powerful idea, where knowledge from one NN can be transferred to another.

Further, the topic of NN is presented and explained with examples. To enable the reader to start programming, needed tools such as Python, Jupyter, Numpy, Pandas and Keras are presented, used and recommended. The first model provides a guide on how to program an NN oneself, and includes the elements; “prepare data”, “initialize weights and biases”, “forward propagation”, “calculating cost”, “backpropagation” and “update parameters”. The second model presents how to program the same model with Keras, an NN framework, also enabling one to build an ML model with few lines of code.

This guide provides examples and tools that can be used for simple demonstrations of ML for regression problems. The thesis provides arguments and justification for the need for further research. It enables a foundation for further development of ML models for different PdM RUL scenarios.

ii. SAMMENDRAG

Et betydelig potensiale og interesse er funnet for Prediktivt Vedlikehold (PdM) og Maskin Læring (ML). Begge feltene er under utvikling og trenger videre forskning. Gjennom samarbeid med det norske vedlikeholdsselskapet, Karsten Moholt, er det funnet to områder av spesiell betydning for denne oppgaven. Det første er å finne mer informasjon om tilnærmingen ML og spesielt potensialet til «Transfer Learning (TL)», eller overførbar læring, og for det andre, å forstå og bygge en ML-modell for PdM.

Denne oppgaven har som mål å gi en introduksjon og være en praktisk startguide for de som er interessert i å bruke ML for PdM. For å oppnå dette er nødvendig bakgrunnsinformasjon samlet, og for å lære mer om ML har flere nettbaserte kurs blitt gjennomført. Nettkurset, bøkene og flere nettguiden er brukt til å bygge og utvikle to ML-modeller for å finne «Remaining Useful Life (RUL)», gjenværende levetid, på «Turbofan Engine Degradation Simulation Data Set» gjort tilgjengelig av NASA. Informasjonen og koden er presentert for å gjøre det mulig for leseren å forstå ML og å gi verktøy for å kunne begynne å bygge ML-modeller.

PdM presenteres som en teknikk for overvåking av driftstilstanden for å gi data som kan sikre maksimalt intervall mellom reparasjoner og minimere antall og kostnader for uforutsette maskinfeil. ML er funnet som et viktig og kraftig verktøy for å finne mønstre og lage spådommer fra en stor mengde data, der «Neural Networks (NN)», nevrane nettverk, er den viktigste teknikken for implementering av ML. TL er funnet som et konsept med stort potensiale, hvor kunnskap fra ett NN kan overføres til ett annet.

Videre er temaet NN presentert og forklart med eksempler. For å gjøre det mulig for leseren å starte programmering, er det presentert nødvendige verktøy som Python, Jupyter, NumPy, Pandas og Keras. Den første modellen gir en veiledning om hvordan man programmerer et NN selv, og inkluderer elementene; “prepare data”, “initialize weights and biases”, “forward propagation”, “calculating cost”, “backpropagation” og “update parameters”. Den andre modellen presenterer hvordan man programmerer samme modell med Keras, et NN-rammeverk, som også gir et eksempel på hvordan man kan bygge en ML-modell med få linjer kode.

Guiden gir eksempler og verktøy som kan brukes til enkle demonstrasjoner av ML for regresjonsproblemer. Master oppgaven gir argumenter og begrunnelser for behovet for videreutvikling av nevnte tema. Oppgaven gir et grunnlag for videreutvikling av ML-modeller for forskjellige PdM RUL-scenarier.

iii. PREFACE

This thesis started with the task of building an Industry 4.0 system for a Predictive Maintenance case. The system was to be based on a developed algorithm, threatening components in a production line differently based on measurements found. When working with the algorithm, the machine learning case came up early as a good solution. Further, Karsten Moholt was contacted to base the system on a real-life case study. Through them, the topic of machine learning became increasingly interesting and significant for this thesis.

I would first and foremost like to thank my supervisor, Professor Amund Skavhaug. I am grateful for his weekly guidance and helpful advice towards this thesis and towards current and future lifestyle. I would also like to thank Stian, Yapi and Ashutosh from Karsten Moholt for their time, interest and valuable help in making this thesis. Last, but not least, I must thank my family and friends for their advice and motivation, especially my dear Lisa Marie, for always being there for me.



Kåre H. Lærum – Trondheim Juli 2018

iv. TASK DESCRIPTION

Title: A study of Machine Learning for Predictive Maintenance - A topic and programming guidance

Task given: February 6th 2018

Deadline: June 26th 2018

Supervisor: Professor Amund Skavhaug, Department of Mechanical and Industrial Engineering, NTNU

Student: Kåre H. Lærum

This Master thesis aims to provide an introduction and to be practical start guide for those interested in using Machine Learning for Predictive Maintenance. To achieve this, firstly, the topics Predictive Maintenance (PdM), Machine Learning (ML) and Transfer Learning (TL) are presented. Then, the thesis aims to provide an understanding towards ML and to give an introduction to the questions; what is ML, how does it work, and how can one build an ML model to start with initial testing?

The candidate shall amongst other:

1. Study necessary background on Predictive Maintenance, Machine- and Transfer learning
2. Build and develop code to find Remaining Useful Life (RUL) on the Turbofan Engine Degradation Data Set provided by NASA
3. Present information and code as an introduction and to be a practical start guide
4. Evaluate the presented guide and the built ML model

V. TABLE OF CONTENTS

i.	Abstract	ii
ii.	Sammendrag	iii
iii.	Preface	iv
iv.	Task Description	i
v.	Table of Contents	ii
vi.	List of Figures	vi
vii.	List of Tables.....	vii
viii.	Abbreviations	viii
1	Introduction.....	ii
1.1	Objectives and Aims.....	ii
1.2	Scope	iii
1.3	Intended Audience	iii
1.4	Report Structure.....	iii
2	Theory	2
2.1	Predictive Maintenance	2
2.1.1	What is PdM.....	2
2.1.2	Preventive Maintenance vs. Predictive Maintenance.....	2
2.1.3	Predictive Maintenance today	2
2.2	Machine Learning.....	3
2.2.1	Machine Learning	3
2.2.2	Machine Learning vs. Deep Learning vs. AI	4
2.2.3	Machine Learning types	5
2.2.4	Machine Learning in Predictive Maintenance	6
2.3	Transfer Learning	7
2.3.1	What is Transfer Learning?.....	7
2.3.2	Transfer Learning challenges:	8
3	Neural Networks	10
3.1	The build of Neural Networks	11
3.1.1	Predicting house prices - A simple Neural Network example	12
3.2	The weights and biases of a neural network.....	13
3.3	Forward propagation.....	14
		ii

3.4	Loss and Cost function	16
3.5	Gradient decent.....	17
3.6	Backpropagation.....	19
3.7	Pre-processing and Data Treatment.....	19
4	Tools for Making a Machine Learning Model.....	20
4.1	Programming Language	20
4.1.1	Python.....	20
4.2	Packages and libraries	20
4.2.1	Anaconda.....	20
4.2.2	Libraries	20
4.3	Machine Learning Frameworks.....	21
4.3.1	Keras.....	21
4.3.2	TensorFlow.....	21
4.3.3	H2O	21
4.4	Data and Training Sets	21
4.4.1	Databases to get datasets for training.....	21
4.4.2	The Turbofan Engine Degradation Simulation Data Set.....	21
5	Building a Machine Learning Model.....	22
5.1	Tools Used.....	23
5.2	Overall Model Description	25
5.3	Prepare data	26
5.3.1	The Turbofan Engine Degradation Simulation Data Set.....	26
5.3.2	Splitting the txt-file and converting to csv-file	27
5.3.3	Load data from csv-file	29
5.3.4	Load N number of motors	31
5.3.5	Standardize values.....	32
5.3.6	Final code for loading treated data.....	33
5.4	Initialize weights and biases	34
5.5	Forward Propagation	36
5.6	Calculate Cost.....	37
5.7	Back Propagation.....	38
5.8	Update parameters	40
5.9	Model 1.....	40

5.10	Running Model 1	42
5.11	Keras	45
5.11.1	Set hyperparameters and load training and test data:	45
5.11.2	Building the Keras model.....	46
6	Discussion	52
6.1	Sources.....	52
6.2	The Potential and Interest of PdM, ML and TL	53
6.3	Understanding Machine Learning	53
6.4	Building and Programming an ML model.....	55
6.5	Usefulness.....	56
7	Conclusion	58
8	Further work.....	60
	References:	62
	APPENDIX A: Turbofan Engine Degradation Simulation Data Set, Readme – From zipfile downloaded from NASA.....	66
	APPENDIX B: Backpropagation - Mathematical calculation of dW and db	68
	APPENDIX C - Code.....	72

APPENDIX A: Turbofan Engine Degradation Simulation Data Set, Readme – From zipfile downloaded from NASA.

APPENDIX B: Backpropagation - Mathematical calculation of dW and db

APPENDIX C: Code

vi. LIST OF FIGURES

Figure 1: AI vs ML vs DL. From Copeland [28]	4
Figure 2: To the left: NN in training. To the right: A trained NN	10
Figure 3: A NN with a single neuron	11
Figure 4: An example of a anN from Nielsen [8]	11
Figure 6: Graphical- and NN representation of predicting price with size. From Ng [6]	12
Figure 7: Drawn NN with several features predicting house price. From Ng [6]	12
Figure 8: NN with several features predicting house price. From Ng [6]	13
Figure 9: NN with three inputs and one neuron. From Ng [6]	14
Figure 10: To the left: Sigmoid activation function. To the right: ReLU activation function. Their respective functions can be found in formula 3.3 and 3.4. From Nielsen [8]	15
Figure 11: Forward Propagation over one neuron. From Ng [6]	15
Figure 12: To the left: Two-layered NN, with four neurons in the hidden layer. To the right: equations z and a in the hidden layer. From Ng [6]	15
Figure 13: Gradient Descent example. From Ng (2017) [6]	17
Figure 14: Visualisation example of $J(w)$. From Ng [6]	18
Figure 15: Steps in a general ML model	22
Figure 16: Steps in a general Keras model	22
Figure 17: Screenshot from Environments in Anaconda Navigator	23
Figure 18: Simple visualisation of the NN	25
Figure 19: Model 1, A two-layered NN, with two neurons in hidden layer and one in the output layer	25
Figure 20: Steps in a basic ML model: Prepare Data	26
Figure 21: The first four and last three rows in Turbofan Engine Degradation Simulation Data Set txt-file FD001	26
Figure 22: Finding corresponding RUL from Cycle Number	27
Figure 23: Folder and file-structure after running after running code to split data	29
Figure 24: The first and last three rows in motor1.csv, from folder train_FD001 in folder train_data. The first row contains headers for each column. By using Pandas DataFrame an unnamed column is automatically added, containing numbers for each row.	29
Figure 25: Elements in a basic ML model: Initialize W and b	34
Figure 26: Steps in a basic ML model: Forward Propagation	36
Figure 27: Steps in a basic ML model: Calculate Cost	37
Figure 28: Steps in a basic ML model: Backpropagation	38
Figure 29: Steps in a basic ML model: Update Parameters	40
Figure 30: Print of cost of Model 1 trained with different hyperparameters	44
Figure 31: Print of cost from Keras model trained with different hyperparameters	49

vii. LIST OF TABLES

Tabell 1: Number of Neurons in each layer in Figure 4.....	11
--	----

viii. ABBREVIATIONS

PdM: Predictive Maintenance

ML: Machine Learning

TL: Transfer Learning

NN: Neural Networks

RUL: Remaining Useful Life

1 INTRODUCTION

The rise of digitization brings with it many different possibilities. One of them is Predictive Maintenance (PdM). Through the use of sensors, changes and faults can be detected, algorithms can translate the changes into recommended actions and machines or humans can take the necessary action to prevent problems, all of which are important elements in PdM. In an article from IIOT World, Saar Yoskovitz, Augury's CEO, mentions connected sensors and Machine Learning (ML) as two main trends taking hold in the PdM space [1].

In a paper by Cline et al. [2], they demonstrated the potential of ML techniques for enhancing the operations of an Oil and Gas equipment service department. In a paper by Li et al. [3], they developed ML techniques for railway PdM. The models are being applied against both historical and real-time data to predict conditions leading to failure.

A Norwegian company implementing ML for PdM is Karsten Moholt. They are one of the world leaders in electromechanical machinery and has over 72 years of combined experience with service, maintenance and condition monitoring. With their PdM program, skAIwatch, Karsten Moholt intends to provide the customers with an end to end solution from detection of potential failure to smart maintenance [4]. Working with Karsten Moholt two areas of special significance for this thesis were found. Firstly, finding more information regarding the ML approach and especially the potential of Transfer Learning (TL). Secondly, understanding and building an ML model for PdM.

In their book, "Building Machine Learning Systems with Python", Coelho and Richert [5] explain the "how" behind the machine learning model. Through their efforts in learning about ML both Authors experienced that much of the information behind ML was "black art", and not usually taught in standard textbooks.

Several books, guides, courses and webpages are made to help others get started with ML. Some of them are the course by Ng [6], a webpage by Brownlee [7] and books from Nilsen [8] and Coelho and Richert [5]. But, few of them are focusing on regression and other practical ML implementations for PdM.

1.1 OBJECTIVES AND AIMS

This Master thesis aims to provide an introduction and to be practical start guide for those interested in using Machine Learning for Predictive Maintenance. To achieve this, firstly, the topics Predictive Maintenance (PdM), Machine Learning (ML) and Transfer Learning (TL) are presented. Then, the thesis aims to provide an understanding towards ML and to give an introduction to the questions; what is ML, how does it work, and how can one build an ML model to start with initial testing?

1.2 SCOPE

This thesis focuses on providing information for the topics PdM, ML and TL. The ML system presented is mainly focused on supervised learning. The guide and code are presented in the light of a RUL (Remaining Useful Life) regression problem, using the Turbofan Degradation Data Set by NASA, where the aim is to predict remaining cycles until failure. The process is explained by both theory and code. One ML model is programmed to show how to build a Neural Networks (NN) oneself and another is programmed to show how one can build an NN with Keras, a neural networks API. The code is not directly optimized for the dataset, but instead, intend to demonstrate how to build an ML model to get initial results and thus to start with initial testing.

1.3 INTENDED AUDIENCE

This thesis is mainly intended for students, and others, with a technical background, who are interested in using supervised ML for regression problems connected to RUL and PdM. It is assumed that the reader has little experience with ML. Some, but not excessive knowledge of programming is assumed.

1.4 REPORT STRUCTURE

Chapter 1 – Introduction

Chapter 2 – Theory: Predictive Maintenance, Machine Learning and Transfer Learning

This chapter present the general theory behind each topic. It focuses on the status of each topic today, along with their possibilities and challenges. The aim is to present the general characteristics of each topic and to provide arguments for the case of PdM, ML and TL.

Chapter 3 –The Neural Network This chapter presents the theory behind the build and components of an NN, and thus ML. The theory focuses on NN used for regression. It aims to answer the question of what ML is, and how ML works, by providing background information and examples for understanding how NN work, and by presenting the general elements to consider when building an ML model.

Chapter 4 – Tools for making a Machine Learning model This chapter present programs, libraries, tools and elements that are used and recommended for building the ML Model. Among them are the programming language Python, the mathematical library Numpy, the data structure library Pandas, and Keras, a high-level NN API. The aim of this chapter is to present tools that are necessary and helpful and when building a neural network.

Chapter 5 – Building a Machine Learning Model This chapter presents the process of building and programming an ML model with python. Firstly, the process, along with code, for editing and prepare the data is presented, then a basic ML model, and lastly a model built using Keras. Both models are built to find RUL form the Turbofan Degradation Dataset.

Chapter 6 – Discussion This chapter discusses topics and elements presented in the thesis. It discusses the three topics, PdM, ML and TL, the code and the guide solution presented.

Chapter 7 – Conclusion

References: The references used in the master thesis.

APPENDIX A: Turbofan Engine Degradation Simulation Data Set, Readme – From zip-file downloaded from NASA.

APPENDIX B: Backpropagation - Mathematical calculation of dW and db

APPENDIX C: Code

2 THEORY

This chapter presents the general theory behind Predictive Maintenance (TL), Machine Learning (ML) and Transfer Learning (TL). It focuses on the status of each topic today, along with their possibilities and challenges. The aim is to present the general characteristics of each topic and to provide arguments for the case of PdM, ML and TL.

2.1 PREDICTIVE MAINTENANCE

2.1.1 What is PdM

Predictive Maintenance (PdM) has many definitions. In their article, Cheng et al. [9] state that the goal of Predictive Maintenance is to save money and increase equipment reliability. Amruthnath and Gupte[10] state that the main purpose of PdM is to reduce unscheduled downtime and consequently improve productivity and reduce production cost. The book, “An introduction to predictive maintenance” by Mobley[11], present several more definitions. PdM can be summarized to be a technique for monitoring operating condition to provide data that can ensure the maximum interval between repairs and minimize the number and cost of unscheduled outages created by machine failures.

2.1.2 Preventive Maintenance vs. Predictive Maintenance

Sciban [12] states in his article the importance of distinguishing between predictive and preventative maintenance as then often can be mixed.

All **Preventive Maintenance** management programs are time-driven. Machine repairs or rebuilds are scheduled based on the MTTF statistics Mobley [11]. In other words, preventive maintenance seeks to decrease the likelihood of a machine’s failure through the performance of regular maintenance. **Predictive Maintenance** relies on data to determine a machine’s likelihood of failure before that failure occurs. The manufacturer can then use a policy to predict and fix. [12]

2.1.3 Predictive Maintenance today

In a report from IoT Analytics, a leading provider of market insights for Internet of Things [13] [1], it is found that maintenance strategies move from Condition-based Maintenance to Analytics-and IoT-enabled PdM [13] [1]. The report forecasts an annual growth rate for PdM of 39% between 2016-2022, and with annual technology spending reaching almost 11 Billion dollars by 2022.

In his article, Sciban [12] , present two key reasons for rise of PdM. The first reason being that modern machinery often comes with embedded computer chips for reading and control, enabling a potential for data capture. Secondly, that the cost of implementing embedded sensors and other new information technologies has and continues to be significantly reduced.

As mentioned, PdM is a process to ensure the maximum interval between repairs and minimize the number and cost of unscheduled outages created by machine failures. Going into detail, PdM techniques help operators and technicians to characterize the condition of equipment and give them warnings and alerts automatically in case of potential problems. Through their “Call for papers”, IEEE [14] mention several benefits. Being aware of which equipment will need maintenance, "unplanned stops" can turn into shorter and fewer "planned stops", giving a benefit of increased availability. Other benefits mentioned is increased equipment lifetime, plant safety, a decrease of accidents and, as a consequence, of the negative impact on the environment, as well as optimized spare parts handling. Through their study, Mulders and Haarman [15] surveyed 280 companies in Belgium, Germany and the Netherlands. They have found few at a level where they are fully implementing PdM, but many that are interested and ambitious to use and improve their PdM solution.

From an article by Nowitz [16] has found most Predictive Maintenance techniques not scalable in to must facilities. Although there are numerous Predictive Maintenance solutions with varying costs and levels of effectiveness, there is no one solution that applies to an entire facility. A further limiting factor mentioned by Mowitz [16] the access to data. There are often third-party machine vendors or other factors restricting operational access to the data. In their study Mulders and Haarman [15] has also challenges connected to few reference cases, data network capacity, and hazardous industrial environments demand an IoT to mention some.

2.2 MACHINE LEARNING

2.2.1 Machine Learning

In his article Kelly [17] lists three big breakthroughs that have greatly affected the ML evolution; Cheap parallel computing - GPUs, Big Data, and better algorithms. Through his online course at Coursera, Ng[6] mentions similar reasons. He believes Deep Learning is taking off due to digitization, the large amount of data available, and the development of Neural Networks (NN). There is a fast evolution of research areas affected by machine learning with close-to-weekly publications of research [18]. The vast collection of data containing images, texts, videos, location, social network activities and more, has given the opportunity for computers to look for patterns and structures that could not be found in smaller data sets. Deep Learning is one of the new methods of finding structures and patterns through multiple levels of abstractions [19], and is currently used in most research involving machine learning [20]. ML and NN can compute any possible function [8]. In an article on Forbes, Marr [21] presents 27 examples of ML in practice. ML is helping in finance, healthcare, energy and manufacturing to mention some. ML has also a significant potential for PdM [15, 16, 22, 23].

In the Crowdfunder’s 2017 Data Scientist report [24] participants where asked to identify the biggest bottleneck in successfully completing AI projects. Over half the respondents named issues related to training data such as *“Getting good quality training data or improving the training dataset”*.

2.2.2 Machine Learning vs. Deep Learning vs. AI

Machine learning is a subsection, or sub-discipline, of Artificial Intelligence (AI), and Deep Learning a subsection of ML [10, 25-28].

In his article Marr [25] also want to present ML as the current state-of-the-art in the field of AI, and the technique is showing the most promise at providing tools that industry and society can use to drive change. In turn, Deep Learning is the cutting-edge of ML. In other words, Deep Learning is the most common technique in ML, and ML the most common technique in AI.

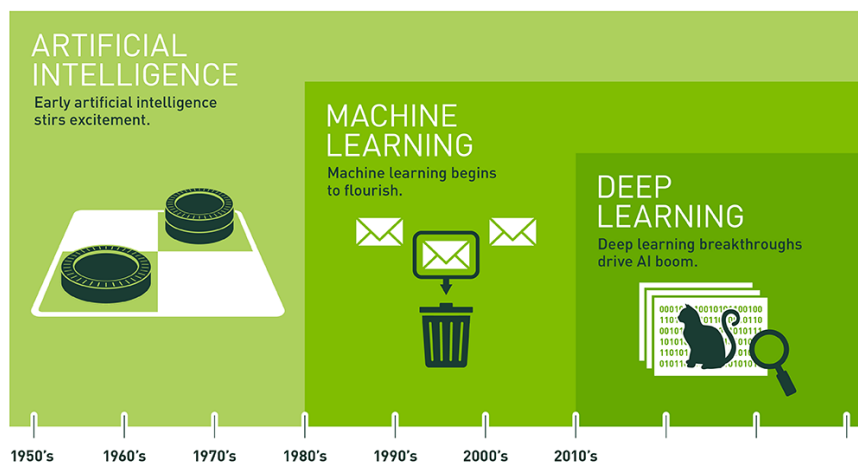


Figure 1: AI vs ML vs DL. From Copeland [28]

2.2.2.1 AI

AI was first introduced by a handful of computer scientists at the Dartmouth Conferences in 1956. The aim was to construct complex machines that possessed the same characteristics of human intelligence. This is today the concept of “General AI”, machines that have all of human senses, reason, and intellect. [28]

2.2.2.2 Machine Learning — An Approach to Achieve Artificial Intelligence

Machine Learning at its most basic the practice of using algorithms to parse data, learn from it, and then decide or predict. Opposite from hand-coding software routines with a specific set of instructions to then accomplish a particular task, the machine is “trained” using large amounts of data and algorithms that give it the ability to learn the correlation between input and desired output and thus how to perform the task. [28, 29]

2.2.2.3 Deep Learning — A Technique for Implementing Machine Learning

Deep Learning can be presented as ML on a “deeper” level and is built on an algorithmic approach called Artificial Neural Networks. Neural Networks (NN) are inspired by our understanding of the biology of our brains and built with layers, connections, neurons, and directions of data propagation. [6, 28, 29]

In 2012, Andrew Ng, at Google, took the NN drastically increased their size. He increased the layers and the neurons, and then run massive amounts of data through the system to train it, images from 10 million YouTube videos. Copeland [28] credits Ng with putting the “deep” in deep learning, with deep describing all the layers in a deep NN.

Through his course, Ng [6] presents and talk about NN, Deep Learning and ML practically the same thing.

2.2.3 Machine Learning types

Learning in ML refers to the process of describing or modeling the available data. ML is mainly divided into three distinct types of learning; supervised-, unsupervised- and reinforcement learning.

2.2.3.1 Supervised Learning

Definition from Encyclopaedia of the Sciences of Learning [30]

“Supervised Learning is a machine learning paradigm for acquiring the input-output relationship information of a system based on a given set of paired input-output training samples. As the output is regarded as the label of the input data or the supervision, an input-output training sample is also called labeled training data, or supervised data.”

Supervised learning problems are generally categorized into "regression" and "classification" problems. In a regression problem, the aim is to predict results within a continuous output or to map input variables to some continuous function. In a classification problem, the aim is instead to predict results to a discrete output, or in other words, map input variables into discrete categories. [6]

There are different types of neural network. One example is Convolution Neural Network (CNN) often used for image application. Another is Recurrent Neural Network (RNN). Often used for one-dimensional sequence data such as translating English to Chinses or a temporal component such as text transcript. As for the autonomous driving, it is a hybrid neural network architecture. Almost all the economic value created by neural networks has been through supervised learning. [6]

2.2.3.2 Unsupervised learning

Definition from Encyclopaedia of the Sciences of Learning [31]

“Unsupervised learning is when a model or system is not supplied with any explicit feedback. The system has to learn patterns or structures in the data on its own.”

Unsupervised learning problems can be further grouped into clustering and association problems [32]. **Clustering** is the process of finding the inherent groupings in the data, an example being customers grouped by purchasing behavior. For **association** the aim is to find rules that describe large portions of the data, an example being people that buy X also tend to buy Y.

A comment from an interview with Geoffrey Hinton [6]

“... in the long run, I think unsupervised learning is going to be absolutely crucial. But you have to sort of face reality. And what's worked over the last ten years or so is supervised learning.”

2.2.3.3 Reinforcement learning**From Encyclopaedia of the Sciences of Learning [33]**

“Reinforcement learners interact with their environment and use their experience to choose or avoid certain actions based on their consequences. Actions that led to high rewards in a certain situation tend to be repeated whenever the same situation recurs, whereas choices that led to comparatively lower rewards tend to be avoided.”

The essence of Reinforcement Learning is learning through interaction. Practically this is implemented by setting the machine in a certain state, then have it take an action, bringing it to another state. Further, the essence of Reinforcement Learning is to define the final aim or final desired state. If the action the system took brings it nearer the final state, it is stored as a positive action, if it brings it further away, it is stored as a negative action. [34]

Another learning type often mentioned is **semi-supervised learning**. It is a hybrid of supervised and unsupervised learning where some data is labeled. Obtaining labeled data for supervised learning can be costly, but often large amounts of unlabeled data can be obtained cheaply. Semi-supervised learning exploits both at once and is useful when only limited labeled data is available. [35]

2.2.4 Machine Learning in Predictive Maintenance

Irwin, MSV and Nowitz [16, 22, 23] all argue ML to be highly relevant for PdM. Irwin argues PdM as one of the most relevant areas where ML can be implemented in the industrial sector. Nowitz [16] states PdM as one of the biggest opportunities identified for the Smart Factory. With ML algorithms asset degradation and breakdowns can be detected ahead of time, and resources can be shifted away from unnecessary maintenance.

Through their study for PwC and Mainnovation, Mulder and Haarman [15] mention the large amount of data as both a challenge and an opportunity. For humans, the data amount could easily be overwhelming. Advancement in ML algorithms is found to be particularly crucial to make use of the data and to make models to improve the case of maintenance. Every subsequent amount of data is then used to refine that model and improve its predictive powers.

Building on this, Cline et al. [2] have through their study found that ML provides a complementary approach to maintenance planning. By analysing significant datasets of individual machine performance and environment variables, it can identify failure signatures and profiles, and provide an actionable prediction of failure for individual parts.

In another study, Susto [36] proposes an ML system for the case of PdM. As new information is available for processed components, the system uses ML and regularized regression methods, to refine Remaining Useful Life estimates (RUL) and associated costs.

In their Call for papers, the 15th IEEE International Conference on Networking, Sensing and Control [14], presents a need for papers on PdM and ML. They wish to identify challenges related to the application of ML techniques to PdM systems.

2.3 TRANSFER LEARNING

2.3.1 What is Transfer Learning?

From Ng[37]

“One of the most powerful ideas in deep learning is that you can take knowledge the neural network has learned from one task and apply that knowledge to a separate task”

Definition from Encyclopaedia of the Sciences of Learning [38]

“Transfer refers to the influence of earlier learning on later learning. Some kinds of transfer take the form of simple stimulus generalization, while in more complex learning situations transfer may depend on the acquisition of rules or principles that apply to a variety of different circumstances. Learning can be viewed as intermediate between simple generalization and the more complex transfer phenomena involved in hierarchically organized skills. “

An example of TL in ML can be found in the article by Masashi, and Pan and Yang [39, 40]. Given the scenario of predicting overall positivity or negativity, sentiment analysis, of digital camera reviews, but there are too few reviews to train on. In this case, there exists an abundance of labeled data from food reviews. By using food reviews as input for training and predict results on digital cameras, the predictions will most likely be poor. With TL, the idea is to “transfer” the similarities, or characteristics, from the scenario with abundant of examples to the scenario with few examples. Both reviews can be said to be written in textual form using the same language, and they both express views about a purchased product. Transferring these characteristics and training the Neural Network on the digital camera reviews can improve the results [40].

Through his webpage, the Ph.D. student Ruder [41] describes TL as the next frontier in ML for the industry, and a solution to mitigate several challenges found for ML in the industry, one of them being the case where machines often meet a different environment in the field than during testing.

An example of TL in PdM can be found at the Maintenance Company Mtell [42]: They use TL to find signature features of a type of machine. By transferring these signatures this they can re-train the algorithm, or ML model, to fit the new motors specific normal and failure signatures.

2.3.2 Transfer Learning challenges:

For TL, Pan and Yang [40] present three main research issues: what to transfer, how to transfer, and when to transfer.

What to transfer asks which part of knowledge or information can be transferred across domains or tasks. Some knowledge is specific for an individual domain, other knowledge may be common between different domains. Such common knowledge may improve the performance of the target domain or task.

When to transfer asks in which situations transferring should be done, or not done. In some situations, when the source domain and target domain are not related to each other, they have no common “knowledge”, transfer degrades the performance of learning in the target domain. This situation is often referred to as negative transfer.

How to transfer relates to the issue of, once after finding when and which knowledge can be transferred, learning algorithms need to be developed to transfer the knowledge.

Avoiding Negative transfer

As one of the main limiting factors of TL, it is important to avoid negative transfer. Pan and Yang [40] suggest studying the transferability between source domains and target domains. Further to define the transferability, a criterion to measure the similarity between domains needs to be defined. Based on the distance measures, one can then cluster domains or tasks, which may help measure transferability, which again can reduce or remove negative transfer.

3 NEURAL NETWORKS

This chapter presents the theory behind the build and components of a Neural Network (NN). The theory focuses on NN used for regression. It aims to answer the question of what ML is, and how ML works. It does this by providing background information and examples for understanding how NN works, and by presenting the general elements to consider when building an NN.

Ng [6] describes NN as powerful learning algorithm inspired by how the brain works. NN can be said to be one of the main drivers behind ML. As previously mentioned, when talking about NN or ML, one is generally talking about the same thing [6].

The goal of an NN is to predict a desired outcome based on input values x . This outcome can be a value or several values. For a regression problem it can be any number, for a classification problem the value, or values, are in-between 0 and 1.

In general, a trained NN can be visualized as a box or system that take input values and output a predicted value, see Figure 2. When training the supervised NN, from a training dataset, the predicted value is measured against the real value. This measure is then used to train and improve the NN. After the NN is trained, a dataset independent of the training set called test set is used to evaluate how good the network is at predicting. With a low score, the process is to go back, edit the NN, train and then test again. Through this process, one can, hopefully, make an NN that is good at predicting the desired outcome. Once a model is sufficiently trained and tested, it can then be used as a prediction application for the given specific area. [6]

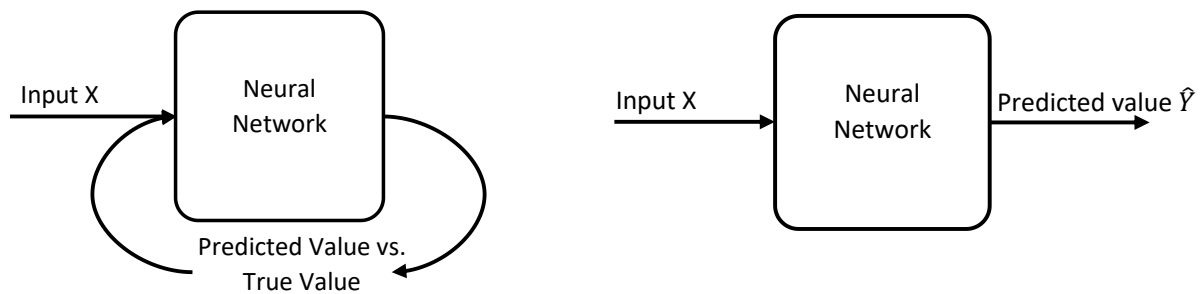


Figure 2: To the left: NN in training. To the right: A trained NN

3.1 THE BUILD OF NEURAL NETWORKS

The simplest NN is just a single neuron, also called node or unit, as seen in Figure 3. A larger NN is then formed by taking many of the single neurons and stacking them together. The process of going from input through the node and to output is similar for all neurons in the network. [6]

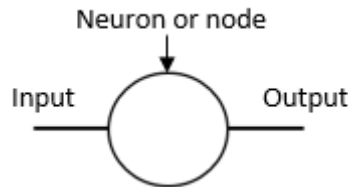


Figure 3: A NN with a single neuron

The general NN has an **input layer**, **hidden layers** and an **output layer**. Each layer has a given number of neurons or nodes in them. Hidden layers are all layers that is not an output or input layer. The output layer corresponds to the predicted values or values. The input is mainly noted as x or X and the output often as \hat{y} or \hat{Y} . [6, 8]

Figure 4 shows an example of 3-layered NN. The input layer is not counted. This 3-layered NN has six neurons in the input layer, four neurons for the first hidden layer, or first layer, three neurons in the second layer and one neuron in the third layer, or output layer.

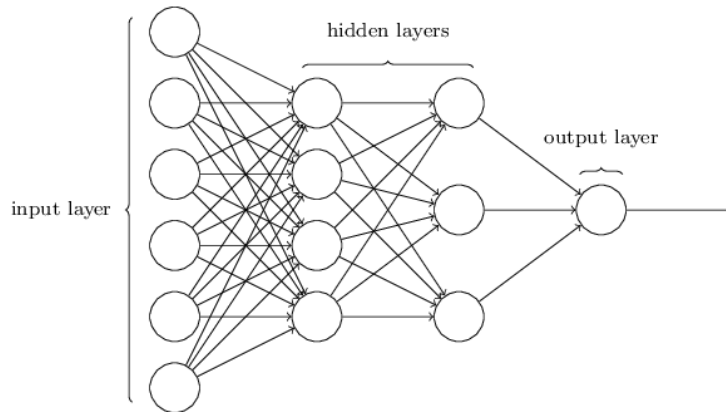


Figure 4: An example of a anN from Nielsen [8]

Input layer	Hidden Layer		Output layer
	First Layer	Second Layer	Third Layer
Six neurons	Four neurons	Three neurons	One neuron

Tabell 1: Number of Neurons in each layer in Figure 4

3.1.1 Predicting house prices - A simple Neural Network example

In the Coursera online course by Andrew Ng, an example is presented to get an intuitive understand about neural networks [6].

Starting simple, the price is only decided by the size of the house. Plotting some values into a graph then, mathematically, the relationship between size of house and price may be estimated as $Price = SomeValue * Size + SomeStartValue$ as seen in left Figure 5. For an NN, this can be illustrated with a single neuron seen to the right in Figure 5. Here x represents the input size, and the predicted output y is the price. As mentioned, the predicted output is normally noted as \hat{y} , but for easier notation it will be noted as y in this example.

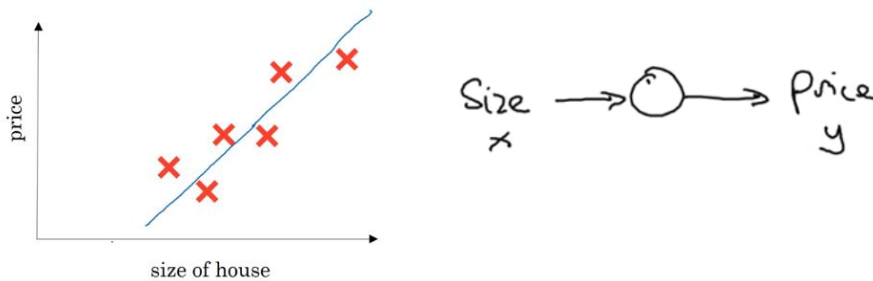


Figure 5: Graphical- and NN representation of predicting price with size. From Ng [6]

To build on this, there may be other features than size that can indirectly affect the price. One of them can be the number of bedrooms. This may be relevant for a family of a given size. Another one can be the zip- or postal code. Depending on location there may be a long or short walk to a potential school, grocery stores or other essential places. A fourth feature can be the general wealth of the neighborhood. This may again effect how good the school is. Based on these features, size and number of bedrooms may say something about the family size. Based on the zip code, walkability may be found. The combination of zip code and wealth may estimate the school quality. For this example, the best estimate for the prize may then be decided by the combination of family size, walkability, and school quality, see Figure 6.

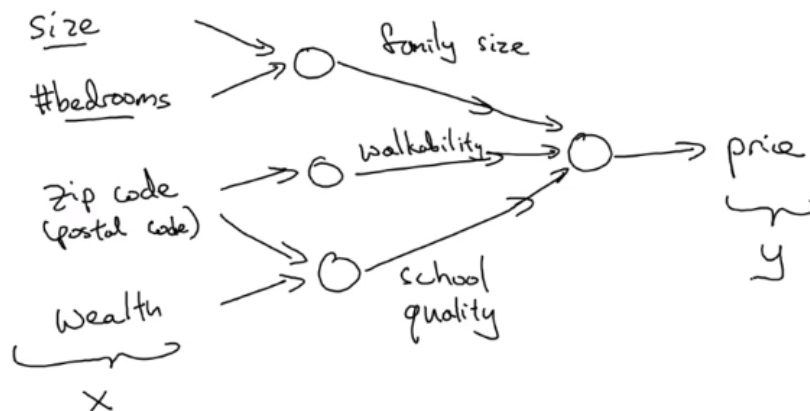


Figure 6: Drawn NN with several features predicting house price. From Ng [6]

To summarize, this example can be described as an NN with four inputs, or features, the size, number of bedrooms, the zip- or postal code, and the wealth of the neighbourhood. Given these input features, the job of the neural network will be to predict the price y . Each of circles drawn in Figure 6 are examples of neurons in the neural network. In the example, the arrows are drawn, and it is already decided which features effects what. When setting up an NN, each of the neurons takes in inputs from all four input features, see Figure 7. Rather than saying the first node represents family size and family size depends only on the features X_1 and X_2 . Instead, the NN will decide.

A comment by Ng (2017) [6]

“the remarkable thing about neural networks is that, given enough data about x and y , given enough training examples with both x and y , neural networks are remarkably good at figuring out functions that accurately map from x to y .”

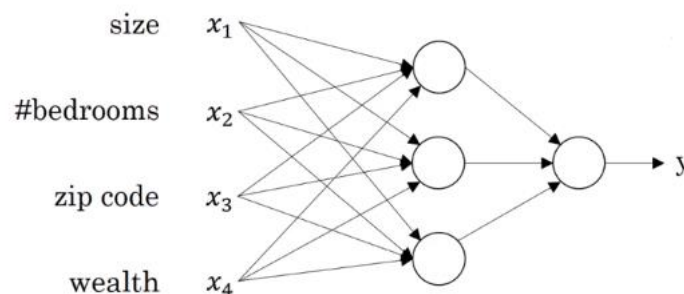


Figure 7: NN with several features predicting house price. From Ng [6]

3.2 THE WEIGHTS AND BIASES OF A NEURAL NETWORK

To explain the thought behind weights and biases, the housing example, Figure 7, is again used. When finding the prize, some of the features may be more relevant than others. Thus, weights (w or W) are introduced. They are real numbers expressing the importance of the change of the respective inputs to the output [8]. Another scenario is when a given feature is important, independent of the feature value. For this scenario bias (b) is added.

For each neuron in the network, the output is pendent on the combination of input values, weights and biases. The mathematical formula with one input over each neuron in the NN network is:

$$y = w * x + b \tag{3.1}$$

When setting up a NN, all weigths and biases in the network is given an initial value. For initial testing, the biases can usually be set to zero and the weights set to a random number [6].

3.3 FORWARD PROPAGATION

Forward propagation is the process of going from input values x , through the NN and to the predicted value \hat{y} . Figure 8 show a scenario with input of one training example, x_1, x_2, x_3 , going through one neuron and giving a predicted output \hat{y} .

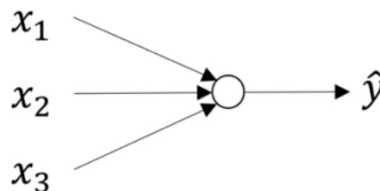


Figure 8: NN with three inputs and one neuron. From Ng [6]

With $x = x_1, x_2, x_3$, forward propagation can be written as

$$\hat{y} = w * x + b \quad (3.2)$$

Where w is a vector containing w_1, w_2, w_3 each a value corresponding to x . Usually, the value found in each neuron is noted as z , where the z value of the output layer is equal to the predicted value, \hat{y} .

Activation function

An important implementation in NN is the activation function. The general idea behind it is that the value found in the given neuron, value z , must be higher than a given threshold for the neuron to be activated. If over, the neuron will “fire” and give output one, if lower, the neuron will stay dormant and output zero. For many other NN it is desirable to have the neuron output other values than just zero or one. There are many different activation functions. Two typically mentioned functions are the Sigmoid-, converting z to a number between 0 and 1, and ReLU (Rectified Linear Unit) activation function, outputting the highest of either zero or z . Their equations are presented in (3.3) and (3.4) and visualized in Figure 9.

$$Sigmoid = \frac{1}{1 + e^{-z}} \quad (3.3)$$

$$ReLU = \max(0, w * x + b) = \max(0, z) \quad (3.4)$$

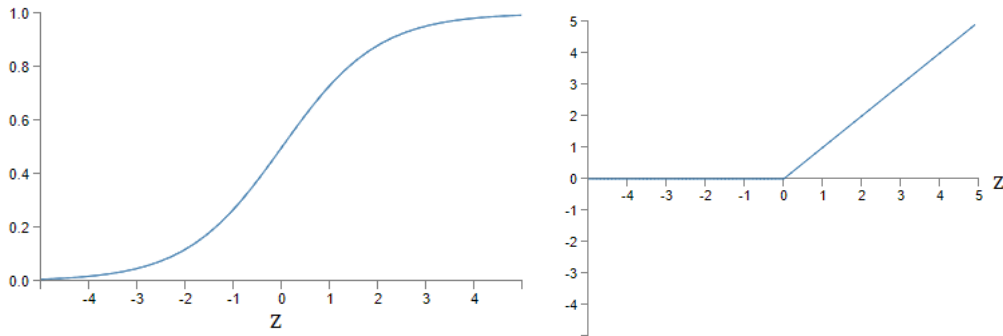


Figure 9: To the left: Sigmoid activation function. To the right: ReLU activation function. Their respective functions can be found in formula 3.3 and 3.4. From Nielsen [8]

Continuing with the example of an NN with one neuron, Figure 8, the general step over one neuron can visualise as seen in Figure 10 and can be written mathematically as (3.5) and (3.6).

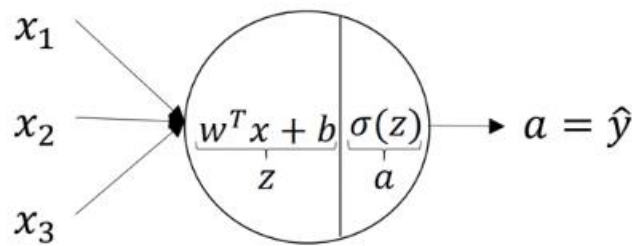


Figure 10: Forward Propagation over one neuron. From Ng [6]

$$z = w^T * x + b \tag{3.5}$$

$$a = \text{Activation function}(z) = \sigma(z) \tag{3.6}$$

To further illustrate the process of forward propagation, a new layer is added to the model, see Figure 11. This new layer contains four neurons. It can now be a challenge to track each function with correct notation. A common notation is to use higher-case for the layer number and lower-case number for each neuron in each layer. For the scenario in Figure 11 the value a found in the first neuron in the first layer is noted $a_1^{[1]}$.

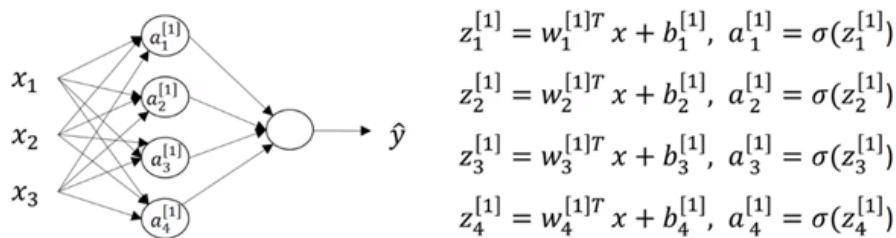


Figure 11: To the left: Two-layered NN, with four neurons in the hidden layer. To the right: equations z and a in the hidden layer. From Ng [6]

Instead of writing the functions a and z for each of neurons, they can be written as a vector for each layer. Giving (3.7) and (3.8) for the first layer and (3.9) and (3.10) for the second.

$$z^{[1]} = W^{[1]} * x + b^{[1]}, \quad (3.7)$$

$$a^{[1]} = \sigma(z^{[1]}) \quad (3.8)$$

$$z^{[2]} = W^{[2]} * a^{[1]} + b^{[1]} \quad (3.9)$$

$$a^{[2]} = \sigma(z^{[2]}) \quad (3.10)$$

From this, the generalized equation over each layer in the NN is noted as (3.11) and (3.12). Where a is the output of one layer and input for the next, and $a^{[0]}$ is equal to x .

$$z^{[l]} = W^{[l]} * a^{[l-1]} + b^{[l]} \quad (3.11)$$

$$a^{[l]} = \sigma(z^{[l]}) \quad (3.12)$$

Until now, all examples are done with one training example. For m training examples the equation over one layer is

$$Z^{[l]} = W^{[l]} * A^{[l-1]} + b^{[l]}, \text{ and } A^{[l]} = \sigma(Z^{[l]}) \quad (3.13)$$

Where each of the parameters $Z^{[l]}$, $A^{[l]}$, $W^{[l]}$ and $b^{[l]}$ are matrices and a vector containing all values in each layer for all training example. As it initially may be more intuitive to think about the parameters only containing the numbers from each example and then loop through the number of examples, later one might see the matrices as more intuitive. When training the NN the process of computing is at least several times quicker when using matrices. [6]

3.4 LOSS AND COST FUNCTION

To train the NN, it is important to define and give a value, or cost, to how good the NN is at predicting the right value. Loss, L , or sometimes called error function, is a function that measure how good our output \hat{y} fit the true label y . One example of a loss function may be the quadratic error function [6].

$$L = (y - \text{predicted})^2 \quad (3.14)$$

Another one logistic loss

$$L = -(y * \log \hat{y} + (1 - y) * \log(1 - \hat{y})) \quad (3.15)$$

The cost function, sometimes noted as J , other times as C , is the average over the sum of the loss function applied to each of the training examples. An example is the quadratic cost function, or also known as mean squared error or MSE.

$$J(w, b) = C(w, b) = \frac{1}{2m} \sum_m (y - \text{predicted})^2 \quad (3.16)$$

When training the ML model, the aim is to minimize the overall cost of the system such that the predicted \hat{y} and true label y are as similar as possible. [6, 8]

3.5 GRADIENT DESCENT

As the aim in training the NN is to minimize the cost function, the aim can also be formulated as to finding the weights and biases which minimizes the cost function, $J(w, b)$. To achieve this, the first step is to find how a change in w and b change the cost. Then we use these changes to update w and b . [6, 8]

A cost function, $J(w, b)$, is shown in Figure 12. The horizontal axes represent the spatial parameters, w and b . In practice, w can be higher dimensions, but for plotting, w and b are represented as single real numbers. The cost function $J(w, b)$ is then, some surface above the horizontal axes. In other words, the height of the surface at a certain point represents the value of $J(w, b)$. The aim thus becomes to find the value of w and b that corresponds to the global minimum of the cost function.[6]

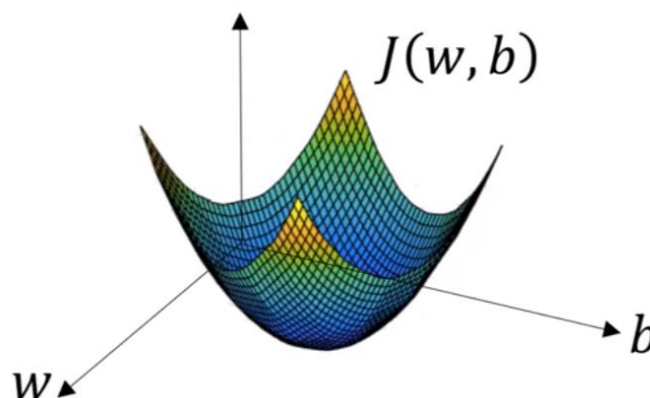


Figure 12: Gradient Descent example. From Ng (2017) [6]

In Figure 12, one can see the cost function J is a convex function, and where the global minimum is. In other scenarios, the cost function can be non-convex and have many different local minimums. For non-convex functions, the global minimum can be hard to find both mathematically and visually. [6]

To find the global minimum, gradient descent starts at an initial point and then takes a step in the steepest downhill direction. After several iterations, one has, hopefully, reached the global minimum. [6, 8]

For further illustration, the cost function is set to be dependent on w , giving $J(w)$, see Figure 13. The first task is to find how much J changes with a change in w , in other words, the slope of the function, $\frac{dJ(w)}{dw}$. Gradient descent does updates w by subtracting the slope multiplied with a given learning rate as see in (3.17). This is done repeatedly until the algorithm converges as illustrated. [6]

$$w = w - LearningRate * \frac{dJ(w)}{dw} \quad (3.17)$$

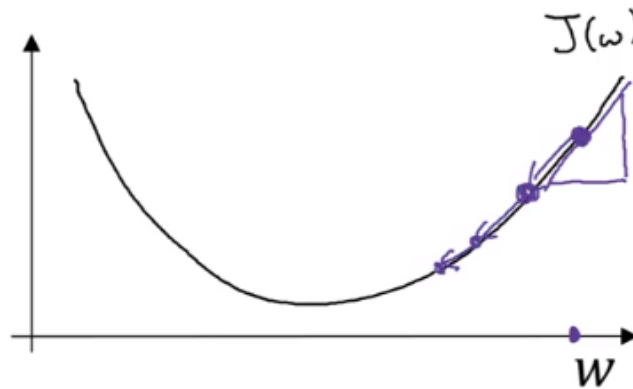


Figure 13: Visualisation example of $J(w)$. From Ng [6]

Initializing parameters

To initialize the dimensions of the parameters W and b correctly, Ng [6] presents (3.18) and (3.19). Where $n^{[l]}$ is the number of neurons in the given layer and $n^{[l-1]}$ is the number of neurons in the previous layer.

$$W^{[l]}: [n^{[l]}, n^{[l-1]}] \quad (3.18)$$

$$b^{[l]}: [n^{[l]}, 1] \quad (3.19)$$

The initialization of values can greatly affect the process of gradient descent and thus the training process of NN. To initialize W with zeros will cause a problem called symmetry breaking problem, resulting in the NN being equivalent to a linear model. A common procedure is to initialize W with random numbers and then multiplying it with a low number such as 0.01. b is usually initialized with zeros, as they do not cause the symmetry breaking problem. The values of initialization are dependent on the system and activation function. An example is the typical sigmoid function where higher values of Z , in other words, higher values of W and b , see (3.9), will result in a slow gradient descent and thus slow learning. [6]

3.6 BACKPROPAGATION

Gradient decent illustrate how the partial derivatives $\frac{dJ}{dW}$ and $\frac{dJ}{db}$ can be used to find the minimized cost. Backpropagation is the process to compute the partial derivatives $\frac{dJ}{dW}$ and $\frac{dJ}{db}$ of the cost function [6, 8]. The general process is to use the chain rule to derivate over each layer.

In his online book Nielsen [8] warn readers

“Be warned, though: you shouldn't expect to instantaneously assimilate the equations. Such an expectation will lead to disappointment. In fact, the backpropagation equations are so rich that understanding them well requires considerable time and patience as you gradually delve deeper into the equations.”

3.7 PRE-PROCESSING AND DATA TREATMENT

The general state of the data available may not be formatted and ready for the ML model. The data thus often need treatment. In his article, Tunkelang [43] states that ML is only as good as the data given to it. Both Brownlee [44] and Ng [6] states the importance of data-treatment for ML. To methods of data-treatment is feature scaling, and feature selection.

Feature scaling through standardization can be an important step for data pre-processing and for the ML model especially if there is a high variance between the features. Standardization is the process of rescaling the features such that all values have a mean of zero and unit variance. Standardization is a pre-processing step that almost always improves the ML system. [6, 45, 46]

Feature selection is a data pre-processing strategy. Li et al. [47] state feature selection as essential to datamining and ML applications and a proven method that is effective and efficient in preparing high-dimensional data. The objectives of feature selection include; making models simpler and more comprehensible, improving the performance of data mining, and preparing clean and understandable data.

4 TOOLS FOR MAKING A MACHINE LEARNING MODEL

This chapter present programs, libraries, tools and elements that is used and recommended for building an ML Model. The aim of this chapter is to give a brief presentation of each tool and its use.

4.1 PROGRAMMING LANGUAGE

4.1.1 Python

Coelho and Richert [5] present ML and Python as a dream team. They state the ML approach as an iterative process, and that it is exactly this that makes Python such a good language for ML. Python is an open-source language that is widely used in the industry or for academical purposes. Python has several useful libraries for easier operations such as NumPy, Pandas, Sklearn and SciPy. It also has several deep learning frameworks that run on top of Python, such as Tensorflow, Keras, PaddlePaddle. [6]

Jupyter

Jupyter Notebook App is a notebook editing and running python documents via a web browser [48]. Through their webpage, a course [49] describes Jupyter as a tool where one can interactively work with code. Jupyter notebook is also used through Ng's [6] online course. With Jupyter's cell structure it enables one to run one cell at a time, enabling quick testing and prototyping.

4.2 PACKAGES AND LIBRARIES

4.2.1 Anaconda

Anaconda Distribution includes 250+ popular data science packages. Anaconda enables the possibility to install, run, and upgrade data science and ML environments and libraries like Scikit-learn, TensorFlow, Keras, and Numpy [50]. **Anaconda Navigator** is a desktop graphical user interface that allows launch of applications and management of packages, environments and libraries [51]. Guide for download is found through their webpage found through the reference.

4.2.2 Libraries

In Python, **NumPy** and **SciPy** enable off-load number crunching tasks to the lower layer in the form of C or FORTRAN extensions [5]. NumPy provides support of highly optimized multidimensional arrays. SciPy uses those arrays to provide a set of fast numerical recipes. In their book, Coelho and Richert [5] present **matplotlib** as the most convenient and feature-rich library to plot high-quality graphs using Python. **Pandas** [52] provide high-performance, easy-to-use data structures and data analysis tools for the Python programming language [53]. Pandas offer matrix visualization matrix treatment. **Sklearn** is the python name for **Scikit-learn**. Scikit-learn is built on NumPy, SciPy and matplotlib. It is built as an efficient and simple tool for handling data, among them data pre-processing and analysis [54].

4.3 MACHINE LEARNING FRAMEWORKS

4.3.1 Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation with easy and fast prototyping. Being able to go from idea to result with the least possible delay. [55]

4.3.2 TensorFlow

TensorFlow is an open-source software library for high-performance numerical computation. Its flexible architecture allows deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. It is originally developed by researchers and engineers from the Google Brain team within Google's AI organization. It comes with strong support for ML. [56]

4.3.3 H2O

On their webpage, H2O.ai describes itself as the leader in AI with its visionary open source platform, H2O. They advertise that more than 12,600 companies use their open-source platform for Finance, Insurance, Healthcare, Retail, Telco, Sales, and Marketing. In February 2018, Gartner named H2O.ai, as a Leader in the 2018 Magic Quadrant for Data Science and Machine Learning Platforms. H2O.ai partners with leading technology companies such as NVIDIA, IBM, AWS, Azure and Google and is proud of its growing customer base which includes Capital One, Progressive Insurance, Comcast, Walgreens and Kaiser Permanente. [57]

4.4 DATA AND TRAINING SETS

4.4.1 Databases to get datasets for training

For training and exercises, it is important to have datasets to train on. Two of them are NASA's Open Data Portal and Kaggle. NASA's data portal has several datasets collected and free to download. The Turbofan Engine Degradation Simulation DataSet is found through NASA [58, 59]. Kaggle's public data platform has several datasets and regular competitions to make the best algorithms on given datasets. [60]

4.4.2 The Turbofan Engine Degradation Simulation Data Set

This dataset is one of the few available datasets that are available for public use. It is made available by NASA through their prognostic data repository [58][59][53], and made by Saxena et al. in 2008 [61]. It is an engine degradation simulation, that was carried out using C-MAPSS tool, containing four different sets that were simulated under different combinations of operational conditions and fault modes. The four set each contains a given number of motors run until failure. The Turbofan dataset can be used to train models to enable better predictions on the PHM08 Challenge Data Set [58].

5 BUILDING A MACHINE LEARNING MODEL

This chapter presents how to build and program two types of python ML models for initial testing. Firstly, code for data-treatment is presented, then a basic ML model and at the end, an ML model using Keras. Both models and the data-treatment are built to predicting Remaining Useful Life (RUL) for the Turbofan Engine Degradation Simulation DataSet from NASA. Mean Squared Error (MSE) and Sigmoid is implemented as cost- and activation functions. The libraries NumPy, Sklearn, Pandas, and matplotlib are used. All code is made and run in the python notebook jupyter.

The general ML model can be said to be composed of six steps, as illustrated in Figure 14. When building a model with Keras, the setup is different, see Figure 15, but it still contains the same elements and ideas. Coelho and Richert [5] state that ML approach is never a waterfall-like process, but a process of going back and forth.

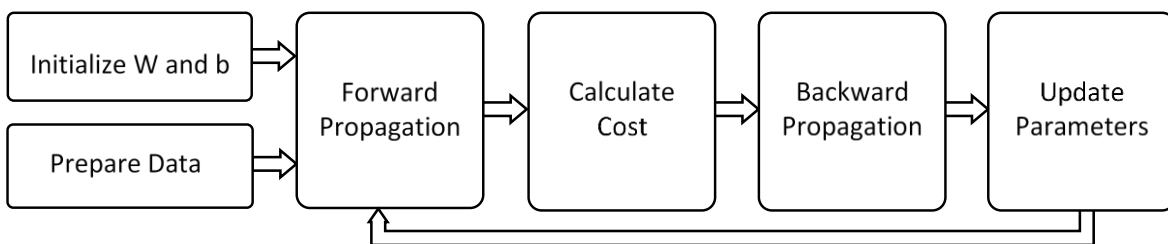


Figure 14: Steps in a general ML model

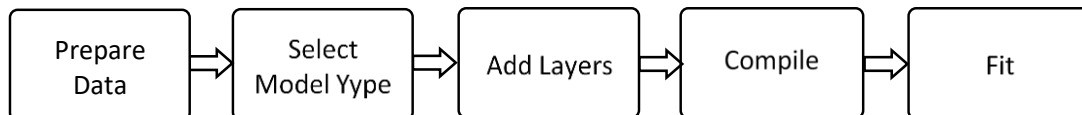


Figure 15: Steps in a general Keras model

When building an ML model, the first aim is to get the model to fit the training set. Once one has sufficient results, one can then test the data on an independent test set. As this guide focuses on the first sub-step of initial testing, code for testing is not presented for Model 1, but can be found in Appendix C. As it is easy to implement and as an example, predicting from a trained Keras model is briefly presented.

A significant part of the code is inspired and guided by other sources. Karsten Moholt have been a part since the beginning. The code is further guided by Ng (2017)[6] through his online course, by Brownlee [7] through his webpage and from Nilsen[8] and Coelho and Richert [5] through their respective books. The documentations from NumPy, Sklearn, Pandas and matplotlib is also used.

All code used can also be found in APPENDIX C

5.1 TOOLS USED

All tools, platforms and libraries used are acquired through Anaconda Navigator. The python notebook Jupyter is installed as a part of the Anaconda Navigator installation. To find and download other libraries, one needs to make and sign into an Anaconda Cloud account. Once this is done, one can start adding libraries, through “Environments”, “base(root)” and then search in the top right corner. When the given library is found, right-click it, or mark it, for installation, see Figure 16.

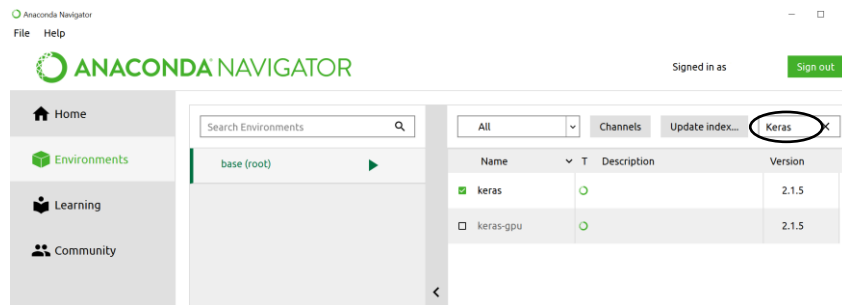


Figure 16: Screenshot from Environments in Anaconda Navigator

For this guide, these libraries are used and installed:

- Numpy
- Pandas
- Sklearn
- Matplotlib
- Keras

Libraries

Through this guide, code from four files are presented. At the beginning of each of these files, relevant libraries are imported. The imported libraries for each file are listed below.

For data-split, txt to csv and creation of file and folders, for file “Split_data”

```
#Import pandas to load and treat data
import pandas as pd

#Import NumPy for array treatment and matrix multiplication
import numpy as np

#Import os for file and folder treatment
import os
```

For data-treatment, for file “Data_treatment”

```
#Import sklearn for mathematical computation
import sklearn
from sklearn import linear_model
from sklearn import preprocessing

#Import NumPy for array treatment and matrix multiplication
import numpy as np

#Import pandas to load and treat data
import pandas as pd
```

To program the basic ML model, for file “NN_2_Layered”

```
#Import numpy for array treatment and matrix multiplication
import numpy as np

#Import sklearn for mathematical computation of MSE of (pred-Y)
import sklearn
from sklearn.metrics import mean_squared_error

#Import matplotlib to visualize results
import matplotlib.pyplot as plt

#Import own code for data-treatment and to load training and test set
from Data_treatment import *
```

To program the Keras Model, for file “Keras”

```
#Import Keras to build and train ML model
from keras.models import Sequential
from keras.layers import Dense

#Import sklearn for mathematical computation of MSE of (pred-Y)
from sklearn import metrics

#Import matplotlib to visualize results
from matplotlib import pyplot as plt

#Import own code for data-treatment and to load training and test set
from Data_treatment import *
```

5.2 OVERALL MODEL DESCRIPTION

The ML model aims to take in setting- and sensor data given from the Turbofan DataSet and predict the RUL in number of cycles left until motor failure. This NN can be visualized in Figure 17.

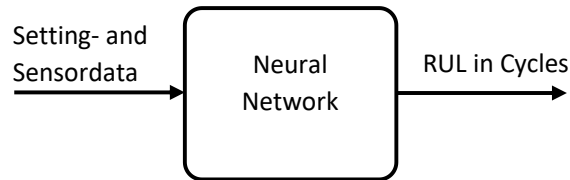


Figure 17: Simple visualisation of the NN

As an initial ML model, the basic model, from here on called Model 1, is built as a two-layered NN, with two neurons in the hidden layer and one neuron in the output layer, giving the NN seen in Figure 18. The activation function Sigmoid is to be used in the hidden layer. As the output is to predict the RUL, no activation function is chosen for the output layer.

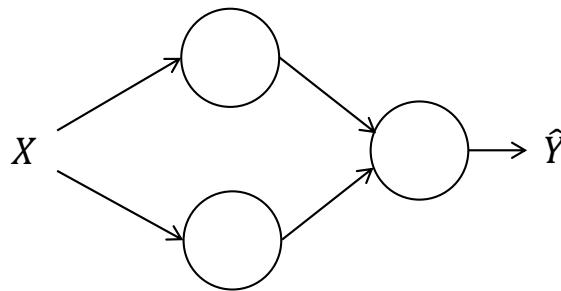


Figure 18: Model 1, A two-layered NN, with two neurons in hidden layer and one in the output layer

5.3 PREPARE DATA

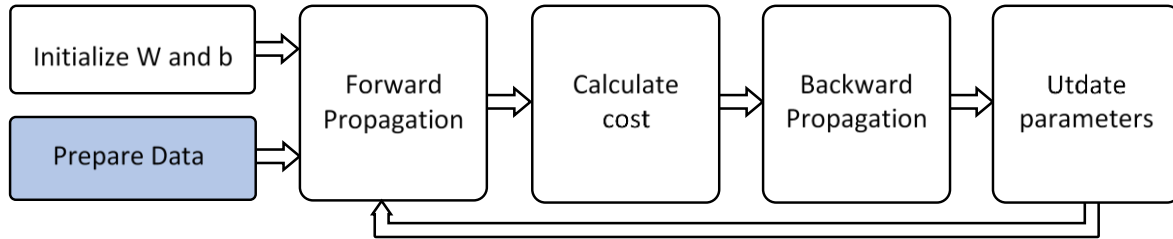


Figure 19: Steps in a basic ML model: Prepare Data

Before building a Supervised ML mode, one need labeled data for training and testing. The data is seldom in a format where it can be directly used by an NN. Thus, the first step is to prepare and edit the data.

The dimension through the network is a common source of error. Thus, one of the initial goals for treating the data is making sure that the dimensions of the input X and labeled values Y are correct. For most NN the input X is of dimensions $[\text{\#features}, \text{\#examples}]$ or its transpose $[\text{\#examples}, \text{\#features}]$, and the labelled values Y of shape $[\text{\#examples}, 1]$. Another element of data-treatment is to make sure that each training or test example has a correct and corresponding Y value. For further treatment, standardizing all values will almost always improve the NN [6].

5.3.1 The Turbofan Engine Degradation Simulation Data Set

The dataset is given as txt-files containing values as seen in Figure 20: The first four and last three rows in Turbofan Engine Degradation Simulation Data Set txt-file FD001. The dataset is further described in sub-chapter 4.4.2 and in APPENDIX A

```

1 1 1 -0.0007 -0.0004 100.0 518.67 641.82 1589.70 1400.60 14.62 21.61 554.36 2388.06 9046.19 1.30 47.47 521.66 2388.02 8138.62 8.4195 0.03
392 2388 100.00 39.06 23.4190
2 1 2 0.0019 -0.0003 100.0 518.67 642.15 1591.82 1403.14 14.62 21.61 553.75 2388.04 9044.07 1.30 47.49 522.28 2388.07 8131.49 8.4318 0.03
392 2388 100.00 39.00 23.4236
3 1 3 -0.0043 0.0003 100.0 518.67 642.35 1587.99 1404.20 14.62 21.61 554.26 2388.08 9052.94 1.30 47.27 522.42 2388.03 8133.23 8.4178 0.03
390 2388 100.00 38.95 23.3442
4 1 4 0.0007 0.0000 100.0 518.67 642.35 1582.79 1401.87 14.62 21.61 554.45 2388.11 9049.48 1.30 47.13 522.86 2388.08 8133.83 8.3682 0.03

20629 100 198 0.0004 0.0000 100.0 518.67 643.42 1602.46 1428.18 14.62 21.61 550.94 2388.24 9065.90 1.30 48.09 520.01 2388.24 8141.05 8.5646
0.03 398 2388 100.00 38.44 22.9333
20630 100 199 -0.0011 0.0003 100.0 518.67 643.23 1605.26 1426.53 14.62 21.61 550.68 2388.25 9073.72 1.30 48.39 519.67 2388.23 8139.29 8.5389
0.03 395 2388 100.00 38.29 23.0640
20631 100 200 -0.0032 -0.0005 100.0 518.67 643.85 1600.38 1432.14 14.62 21.61 550.79 2388.26 9061.48 1.30 48.20 519.30 2388.26 8137.33 8.5036
0.03 396 2388 100.00 38.37 23.0522
20632
    
```

Figure 20: The first four and last three rows in Turbofan Engine Degradation Simulation Data Set txt-file FD001

For this guide, relevant information regarding the dataset is extracted to be:

- The dataset format:
 - Each txt-file has rows representing a training example.
 - Column 1 is “Unit Number”
 - Column 2 is “Cycle”
 - Column 3 to 5 are “Settings”
 - Column 6 to 26 are “Sensor Measurements”

- Both the training- and test set are divided into four txt-files. Each containing a number of motors run.
- Measurements from each motor start with cycle one and measurements are taken for each cycle until failure. The measurements at each cycle corresponds to an example.
- Each example has no direct corresponding RUL. In other words, X has no direct corresponding Y -values.

From this information, some tasks for data treatment can be found. Firstly, loading X from txt-file into the notebook, then to find a corresponding Y for X and, thirdly, delete column “Unit Number” and column 2 “Cycle”, as they are not features related to RUL.

To find the Y corresponding to X , the first number, “unit number”, in the dataset, or txt-file, corresponds to a motor. The “cycle” number always start with number one on each new motor and increases with one for each cycle the motor runs, until failure. Inverting the cycle column for each motor gives each example a corresponding RUL, or remaining cycle, value. To illustrate this, the measurements from the first motor, running 192 cycles, can be used. Extracting the column containing the cycle numbers, then inverting it, the first example, or row, get the corresponding label 192. The second 191, and so on, until the last cycle corresponding to the last example. This is also illustrated in Figure 21.

Unit Number	Cycle Number
1	1
1	2
1	3
⋮	⋮
⋮	⋮
1	192
2	1

Unit Number	Cycle Number
1	192
1	191
1	190
⋮	⋮
⋮	⋮
1	1
2	287

Figure 21: Finding corresponding RUL from Cycle Number

5.3.2 Splitting the txt-file and converting to csv-file

To make the data easier to work with and correctly fit Y to a corresponding X , each of the training and test txt-files FD001 to FD004 are added to their respective folder and then split into a number of different csv-files where each file corresponds to one motor or unit number.

The code below defines two functions `remove_unnamed()` and `split_data()`. Then two arrays containing folder names are created. In the end, a “for loop” goes through each of the folders and with the function `split_data()` add the different csv-files, each containing data and measurements from one motor. As the aim of this code is file and folder treatment, its elements are not explained in detail. The process of loading data is explained later. To run code in Jupyter, click into the cell where the code is, then click shift+enter.

```
#Libraries used
import pandas as pd
import numpy as np
import os

#Function for removing unnamed columns
def remove_unnamed(df):
    return df.loc[:, ~df.columns.str.contains('^Unnamed')]

#Function for splitting txt-file into several csv-files containing a
motor each
def split_data(file_folder):
    data_type = file_folder[0].split('_')[0]
    column1 = ['unit', 'cycle', 'setting1', 'setting2', 'setting3']
    column2 = ['sensor{}'.format(i) for i in range(1,22)]
    cols = column1+column2
    os.makedirs('{}_data'.format(data_type))

    for file in file_folder:
        folder = file.split('.')[0]
        print("creating folder data/{}".format(folder))
        os.makedirs('{}_data/{}'.format(data_type, folder))

    # process files
    for j, file_name in enumerate(file_folder):
        path = "{}".format(file_name)
        data = open(path).readlines()
        my_new_data = list(map(lambda line: line.strip('\n').split(' ')
[: -2], data))

        for k in list(range(1,101)):
            new_array = list(filter(lambda x: x[0]=='{}'.format(k), my_
new_data))
            df = pd.DataFrame(new_array, columns=cols)
            print("processing file {} folder {}_FD00{} mortor{} data"
.format(file_name, data_type, j+1, k))
            df.to_csv('{}_data/{}_FD00{/motor{}.csv'.format(data_type,
data_type, j+1, k))

#Array containing the string names of train and test txt-files
training_files = ['train_FD00{}.txt'.format(k) for k in range(1,5)]
test_files = ['test_FD00{}.txt'.format(k) for k in range(1,5)]

#Run through test and training txt-files to split data
for file_folder in [training_files, test_files]:
    split_data(file_folder)
```

The result of running the code can be seen in Figure 22 and Figure 23. The data is split into folders `test_data` and `train_data`. Both train and test folders are again divided into four folders. In each of these folders, the number of csv-files are divided by the motor unit number. As the data has gone through pandas “DataFrame”, an unnamed index column has been added to each csv-file.

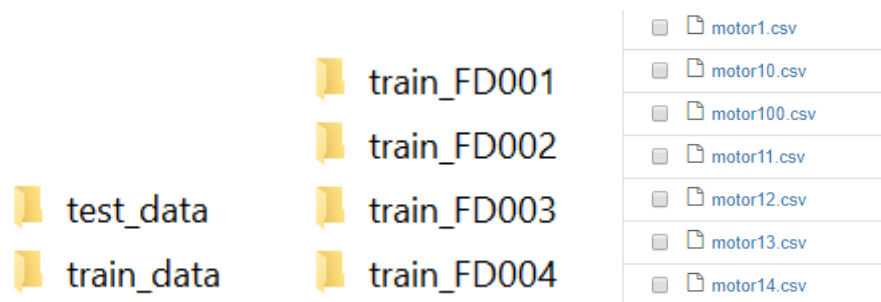


Figure 22: Folder and file-structure after running after running code to split data

1	,unit,cycle,setting1,setting2,setting3,sensor1,sensor2,sensor3,sensor4,sensor5,sensor6,sensor7,sensor8,sensor9,sensor10,sensor11,sensor12,sensor13,sensor14,sensor15,sensor16,sensor17,sensor18,sensor19,sensor20,sensor21
2	0,1,1,-0.0007,-0.0004,100.0,518.67,641.82,1589.70,1400.60,14.62,21.61,554.36,2388.06,9046.19,1.30,47.47,521.66,2388.02,8138.62,8.4195,0.03,392,2388,100.00,39.06,23.4190
3	1,1,2,0.0019,-0.0003,100.0,518.67,642.15,1591.82,1403.14,14.62,21.61,553.75,2388.04,9044.07,1.30,47.49,522.28,2388.07,8131.49,8.4318,0.03,392,2388,100.00,39.00,23.4236
191	189,1,190,-0.0027,0.0001,100.0,518.67,643.64,1599.22,1425.95,14.62,21.61,551.29,2388.29,9040.58,1.30,48.33,520.04,2388.35,8112.58,8.5223,0.03,398,2388,100.00,38.49,23.0675
192	190,1,191,-0.0000,-0.0004,100.0,518.67,643.34,1602.36,1425.77,14.62,21.61,550.92,2388.28,9042.76,1.30,48.15,519.57,2388.30,8114.61,8.5174,0.03,394,2388,100.00,38.45,23.1295
193	191,1,192,0.0009,-0.0000,100.0,518.67,643.54,1601.41,1427.20,14.62,21.61,551.25,2388.32,9033.22,1.30,48.25,520.08,2388.32,8110.93,8.5113,0.03,396,2388,100.00,38.48,22.9649
194	

Figure 23: The first and last three rows in `motor1.csv`, from folder `train_FD001` in folder `train_data`. The first row contains headers for each column. By using Pandas DataFrame an unnamed column is automatically added, containing numbers for each row.

5.3.3 Load data from csv-file

From here on, the rest of the code and functions connected to data-treatment are collect into the python file “Data_treatment.py”.

To run the code in “Data_treatment.py” the libraries listed in the code below are imported.

```
#Import NumPy for array treatment and matrix multiplication
import numpy as np

#Import Sklearn for mathematical computation
import sklearn
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

#Import Pandas to load and for data treatment
import pandas as pd
```


There are several ways to load the dataset from either txt or csv. Through this guide, pandas `pd.read_csv()` is used. After running the code presented in the previous chapter, the dataset is split by commas and the first row contain headers for each column. With this format, `pd.read_csv()` will automatically split values by comma and headers are added to each column.

The following code takes in the string-variable path containing the string “train_data/train_FD001/motor1.csv”. The code returns a matrix X and list Y .

```
#Function to remove the unnamed column added to dataset through the
code for splitting the original dataset.
def remove_unnamed(df):
    return df.loc[:, ~df.columns.str.contains('^Unnamed')]

#Loads data from path. Return Matrix X with shape[#examples, #features]
and a list Y.
def load_data(path):

    #Reads the CSV file and removes unnamed columns
    data = remove_unnamed(pd.read_csv(path))

    #Makes a list with cycle numbers. From 1 to n cycles before failure
    Y = list(list(data['cycle']))

    #Reverse the list to make Y corresponding to RUL for each example
    in X
    Y.reverse()

    #Retrieves values from columns "setting1" to "sensor21".
    Does not retrieve unit number and cycle
    X = data.loc[:, 'setting1':'sensor21'].as_matrix()
    return X, Y
```

Additional explanation:

- **remove_unnamed:** Takes in a dataframe `df` and with `df.loc` runs through all columns and removes columns with header “Unnamed”
- **Path** is set to: “train_data/train_FD001/motor1.csv”
- **pd.read_csv** will automatically split each value in each row into its corresponding columns because each value is split by “,”.
 - **delim_whitespace=True:** Split the values in each row into its corresponding column. Option to be added to `pd.read_csv`.
 - **header=None:** Tells code that there is no header. Without “header” or with “header=True” will set the first row as header.
 - Ex: `pd.read_csv(“data.txt”, delim_whitespace=True, header = None)`
- **Y** is selected as a list because the function `Y.reverse()` and to make it easier to add later motors.

5.3.4 Load N number of motors

In the previous code for loading data only loads data for one motor. For better training, one may want to add more motors. As the input X has to contain all training examples, and Y all correlated label examples, each motor csv-file has to be split, and the respective parts added into the matrix X and list Y . Through this guide, the matrices are of type NumPy array.

In the previously presented code Y was a list with dimensions $[\text{\#examples},]$. To change this into a NumPy array of dimension $[\text{\#examples}, 1]$, the Y data is treated by the function `np.array()` and `np.reshape()`. With NumPy the dimensions are described as shapes. Through `reshape` one can change the dimensions, with `shape` combined with the `print()` function, the dimensions of a given array can be printed. Ex: `print(X.shape)` will print out the dimensions of X .

The following code shows how to load a given number of motors and thus a given number or training examples with corresponding labels. It takes in a given number and returns NumPy arrays X and Y , training and test.

```
#Load n number of motors. Return array X and Y in format
[number of examples, 24], [number of examples, 1]

def load_n_motors(num_motor):

    #Create two arrays containing string names for train and test FD001
    training_folders = ['train_FD00{}'.format(i) for i in range(1,5)]
    test_folders = ['test_FD00{}'.format(i) for i in range(1,5)]

    #Folder number. Here: train_FD001 or test_FD001
    k = 0

    #Make an empty Y train and test list
    Y_train = []
    Y_test = []

    #Load num_motor X and Y training- and test-sets
    for i in range(1, num_motor+1):

        #Create training and test path
        train_path = 'train_data/{}/motor{}.csv'.format(
            training_folders[k],i)

        test_path = 'test_data/{}/motor{}.csv'.format(
            test_folders[k],i)

        #Load temporary X and Y sets
        X_train_temp, Y_train_temp = load_data(train_path)
        X_test_temp, Y_test_temp = load_data(test_path)

        #Build the Y list
        Y_train = Y_train + Y_train_temp
        Y_test = Y_test + Y_test_temp
```

```

#Convert Matrix X to a pandas dataframe
X_train_temp_df = pd.DataFrame(X_train_temp)
X_test_temp_df = pd.DataFrame(X_test_temp)

#For the first loop
if (i == 1):
    X_train_df = X_train_temp_df
    X_test_df = X_test_temp_df

#Append/add new motors to set
else:
    X_test_df = X_test_df.append(X_test_temp_df)
    X_train_df = X_train_df.append(X_train_temp_df)

#Convert X and Y to NumPy arrays
X_train = X_train_df.values
Y_train = np.array(Y_train)
X_test = X_test_df.values
Y_test = np.array(Y_test)

#Reshape Y arrays from [examples, ] to [example1, 1]
Y_train = Y_train.reshape([Y_train[0], 1])
Y_test = Y_test.reshape([Y_test[0], 1])

return X_train, Y_train, X_test, Y_test
    
```

Additional Explanation:

- **training_folders = ['train_FD00{0}'.format(i) for i in range(1,5)]** create an array containing the string names “train_FD001 to train_FD004
- **Append** add values from one “DataFrame” to another
- **Y_train[0] = #examples**

5.3.5 Standardize values

With the data split into desired files and code ready for loading a given number of examples, one could start building the rest for the ML model. Still, some treatment of the values is recommended.

As seen in both Figure 20 and Figure 23, the values between the different settings and sensors have a high variance. As mentioned in sub-chapter 3.7, it is then recommended to standardize the values, rescaling the features such that all values have a mean of zero and unit variance. This can be easily be done through the Sklearn *StandardScaler()* and *fit_transform()*

```

#StandardScaler
def standardize(data):
    d = preprocessing.StandardScaler().fit_transform(data)
    return d
    
```

5.3.6 Final code for loading treated data

The last step is to implement both *load_n_motors()* and *standardize()* together into one function. For easy notation sets for training are noted as *X* and *Y*, and sets for testing are noted as *Xt* and *Yt*. Both the *X* training and test sets are transposed ([#features, #examples] -> [#examples, #features]) for the shape to fit the NN in Model 1.

```
# Load treated data by normalization and robust scaling
def load_treated_data(num_motor):

    #Load train- and testset
    X, Y, Xt, Yt = load_n_motors(num_motor)

    #Standardize values in X and Xt
    X = standardize(X)
    Xt = standardize(X)

    #Transpose X and Xt from [features, examples] to
    [examples, features]
    X = X.T
    Xt = Xt.T

    return X, Y, Xt, Yt
```

5.4 INITIALIZE WEIGHTS AND BIASES

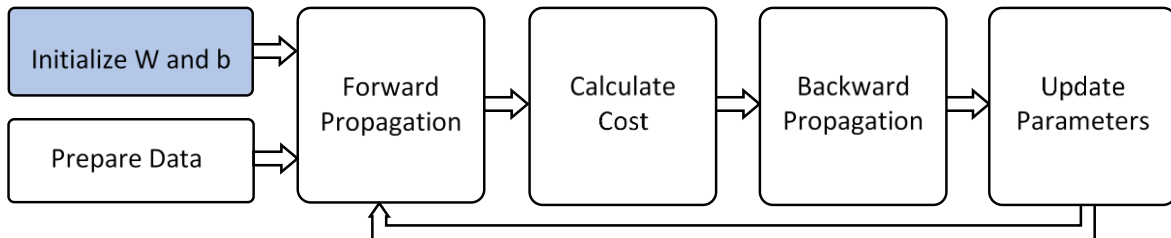


Figure 24: Elements in a basic ML model: Initialize W and b

With the data treated and ready, the next step is to start building the Model 1. All code presented for Model 1 is collected into the file “NN_2_Layer”.

Libraries used:

```

#Import NumPy for array treatment and matrix multiplication
import numpy as np

#Import Sklearn for mathematical computation of MSE
import sklearn
from sklearn.metrics import mean_squared_error

#Import matplotlib to visualize results
import matplotlib.pyplot as plt

#Import own code for data-treatment and to load training and test
set
from Data_treatment import *
  
```

To load functions for loading or treating data, the python file *Data_treatment.py*, made in previous chapters, is imported. A thing to note is, when working in Jupyter, all files are saved as “.ipynb”. To import files, they must be of format “.py”, meaning the “.ipynb” files must be saved as or converted to “.py” files.

As mentioned in the NN chapter, each layer as a set of weights and biases. The collection of all the weights and biases through the NN are the parameters is to be optimized to find the minimized cost function. As mentioned through gradient descent, correct initialization can speed up the process of achieving a minimized cost.

To initialize the parameters, one needs the dimensions of the NN. Here, the function takes in; n_x : input layer, n_h : hidden layer and n_y : output layer, with the corresponding number of neurons in each layer. In the code below the dimensions follow the formulas 3.18 and 3.19 that are given in sub-chapter 3.5. The weights are initialized with random numbers and multiplied with 0.01 as suggested by Ng [6]. The biases are initialized with zeros.

```
#Initialize Parameters W1, b1, W2 and b2 as NumPy arrays
#Input: n_x: #features from training set | n_h: #hidden layers | n_y:
number of nodes in output layer
def initialize_parameters(n_x, n_h, n_y):

    #NumPy array of dimension [n_h, n_x] with random numbers
    W1 = np.random.randn(n_h, n_x)*0.01

    #NumPy array of dimension [n_h, 1] with zeros
    b1 = np.zeros((n_h, 1))

    #NumPy array of dimension [n_y, n_h] with random numbers
    W2 = np.random.randn(n_y, n_h)*0.01

    #NumPy array of dimension [n_y, 1] with zeros
    b2 = np.zeros((n_y,1))

    return W1, b1, W2, b2
```

5.5 FORWARD PROPAGATION

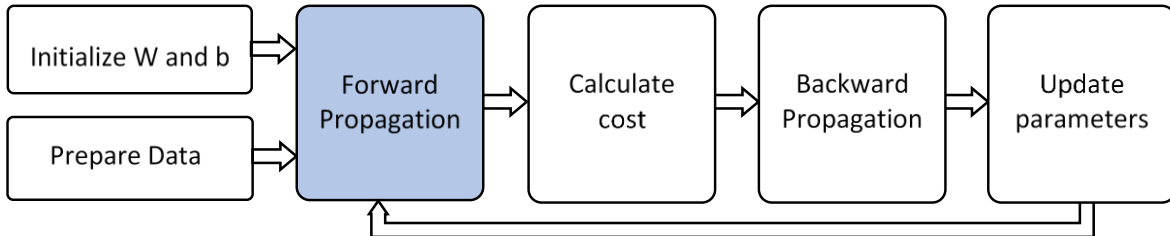


Figure 25: Steps in a basic ML model: Forward Propagation

With W and b initialized, the next step is to enable the process of going from input X to predicted value \hat{Y} . To achieve this one can program the output A from each layer or make a function that enables forward propagation over one layer. In the following code, the latter is chosen and made from the mathematical function 3.13 presented in sub-chapter 3.3. As the NN has Sigmoid activation in the hidden layer and none in the output, the function also takes in a string with either “Sigmoid” or “None” deciding the activation for the given layer.

```

#Forward propagation over one layer: Calculating Z and A(Z)
def forward_prop(W, b, A_prev, a_function):

    # Z = W * A_prev + b
    Z = np.dot(W, A_prev)+b

    if a_function == "Sigmoid":
        A = 1/(1+np.exp(-Z)) #A = Sigmoid(Z)

    elif a_function == "None":
        A = Z #No activation give A = Z
    return A, Z
    
```

Additional Explanation:

$$- \text{np.dot}(W, A_prev) = \sum(w * a_prev) = W * A_prev$$

5.6 CALCULATE COST

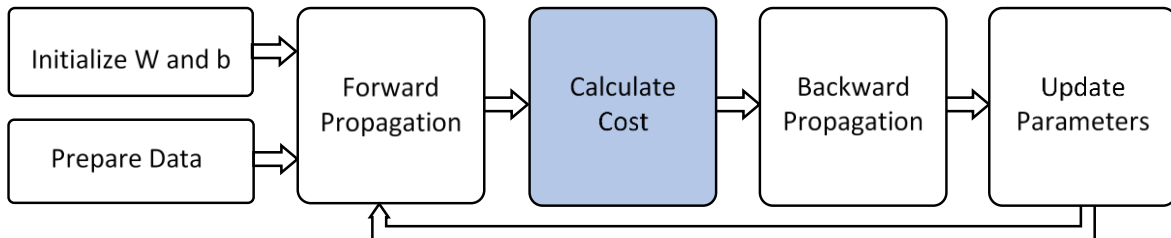


Figure 26: Steps in a basic ML model: Calculate Cost

The Mean Squared Error (MSE) cost function is a common function for regression problems. With two layers, the A matrix found after the second layer, A_2 , is the same as the predicted Y hat (\hat{Y}). Through the calculations set up in `forward_propagation()`, Y hat is of dimensions $[1, \#examples]$, thus to correctly calculate $(Y - Yhat)$, the Y array ($[\#examples, 1]$) is transposed.

```

# Takes in Y and the predicted value from NN and forward propagation,
# and calculates the cost with Mean Squared Error (MSE) cost function
# Both Y and Yhat are NumPy arrays of dimensions [number of examples, 1]

def cost_func(Y, Yhat, m):

    #MSE cost function
    cost = (1/(2*m) * np.sum((Y.T-Yhat)**2))

    # Turns the dimensions of cost from [[17]] into 17).
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    return cost
  
```


5.7 BACK PROPAGATION

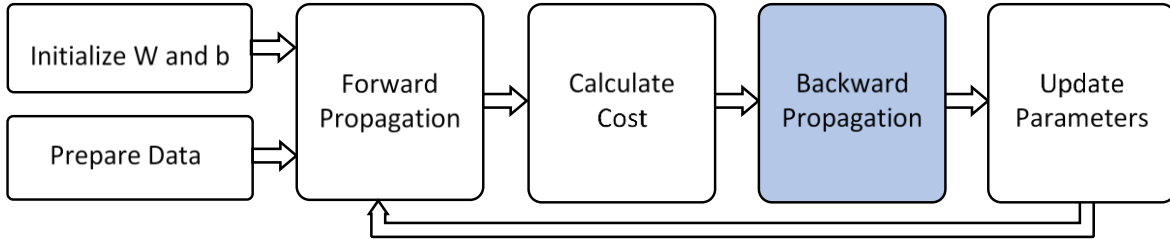


Figure 27: Steps in a basic ML model: Backpropagation

To implement gradient descent and to find the minimized cost, the first step is backpropagation. The process of finding the change in cost by the change in each of its parameters, and the second is to update the parameters through iterations. For the first step, the aim thus becomes to find $\frac{dC}{dW^{[2]}}$, $\frac{dC}{db^{[2]}}$, $\frac{dC}{dW^{[1]}}$ and $\frac{dC}{db^{[1]}}$. Through the use of chain rule from calculus each of these are found from the MSE cost function given as the mathematical formula 3.16 in sub-chapter 3.4. The full calculations can be viewed in APPENDIX B.

The partial derivatives found are

$$\begin{aligned}
 dW2 &= \frac{dC}{dW^{[2]}} = \frac{dC}{dA^{[2]}} * \frac{dA^{[2]}}{dZ^{[2]}} * \frac{dZ^{[2]}}{dW^{[2]}} \\
 &= \frac{1}{m} ((A2 - Y) \circ A1)
 \end{aligned} \tag{5.7}$$

$$\begin{aligned}
 db2 &= \frac{dC}{db2} = \frac{dC}{dA^{[2]}} * \frac{dA^{[2]}}{dZ^{[2]}} * \frac{dZ^{[2]}}{db^{[2]}} \\
 &= \frac{1}{m} \sum((A2 - Y))
 \end{aligned} \tag{5.8}$$

$$\begin{aligned}
 dW1 &= \frac{dC}{dW^1} = \frac{dC}{dA^2} * \frac{dA^2}{dZ^2} * \frac{dZ^2}{dA^1} * \frac{dA^1}{dZ^1} * \frac{dZ^1}{dW1} \\
 &= \frac{1}{m} \sum((A2 - Y) * 1 * W2 * A1 * (1 - A1) * X)
 \end{aligned} \tag{5.9}$$

$$\begin{aligned}
 db1 &= \frac{dC}{db^1} = \frac{dC}{dA^2} * \frac{dA^2}{dZ^2} * \frac{dZ^2}{dA^1} * \frac{dA^1}{dZ^1} * \frac{dZ^1}{db1} \\
 &= \frac{1}{m} \sum(((A2 - Y) * 1 * W2)) * A1 * (1 - A1))
 \end{aligned} \tag{5.10}$$

The sequence and dimensions, transposed or not, of the components used for calculating dW and db are set as they are to enable correct dimensions of dW and db .

```
def backprob(Yhat, Y, A1, X, W2, m):  
  
    #Partial derivatives  
    dCost_dYhat = (Yhat-Y.T)      #dCost/dYhat  
    dYhat_dW2 = A1                 #dYhat/dW2  
    dYhat_dA1 = W2                 #dYhat/dA1  
    dA1_dZ1 = A1*(1-A1)           #dA1/dZ1 = Derivative of sigmoid(Z1)  
    dZ1_dW1 = X                    #dZ1/dW1  
  
    #Calculating dW1 and db1  
    dW2 = (1/m) * np.dot(dCost_dYhat, dYhat_dW2.T)  
    db2 = (1/m) * np.sum(dCost_dYhat, axis=1, keepdims=True)  
  
    #Calculating dW1 and db1  
    dW1 = (1/m) * np.dot((np.dot(dYhat_dA1.T, dCost_dYhat) * dA1_dZ1),  
                          X.T)  
    db1 = (1/m) * np.sum((np.dot(dYhat_dA1.T, dCost_dYhat) * dA1_dZ1),  
                          axis=1, keepdims=True)  
  
    return dW1, db1, dW2, db2
```

Additional Explanation:

- **np.sum:** To keep the correct dimensions of db the elements axis=1 and keepdim=True are added.

5.8 UPDATE PARAMETERS

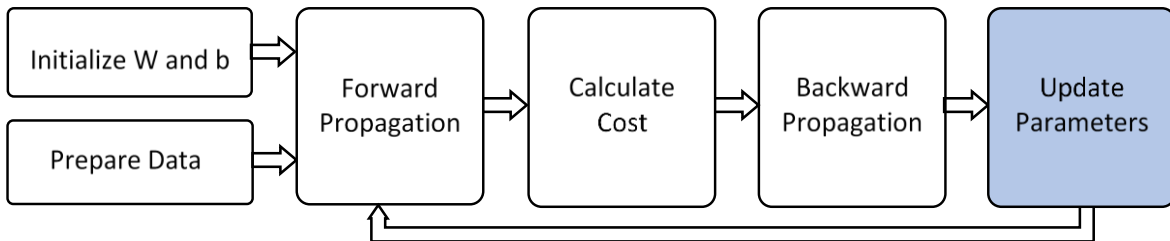


Figure 28: Steps in a basic ML model: Update Parameters

The last step of gradient descent is to update the parameters. Differently from the previously presented code, this element is not written as a function but directly implemented into the final model code. Through each iteration, the parameters are updated.

```

#Update Parameters
W1 = W1 - learning_rate *dW1
W2 = W2 - learning_rate *dW2

b1 = b1 - learning_rate *db1
b2 = b2 - learning_rate *db2
    
```

5.9 MODEL 1

Finally, all steps can be implemented to build the ML model. This model takes in the datasets X and Y , and the hyperparameters $learning_rate$ and $epoch$, where $epoch$ corresponds to the number of iterations, or training cycles. Hyperparameters are parameters one can manually change to try to improve how well the built ML model predicts [6]. The model returns the final parameters, W and b , found through training. These parameters can then be used for testing and later as a part a finished ML model.

Other elements implemented are n_x , n_h and n_y . An alternative could be to take them in as an input, as one may want to change the number of neurons in the hidden layer. m is defined as the number of training examples. cp is an array defined to store the found cost in each iteration. It is defined to present a model of the cost.

```

# Model 1
def model(X, Y, learning_rate, epoch):

    # Set the number of neurons in each layer
    n_x = X.shape[0] #Input layer
    n_h = 2          #Hidden layer
    n_y = 1          #Output layer

    m = Y.shape[0] #Number of examples
    cp = np.zeros((epoch-1)) #Array for storing cost in each epoch
    
```

```

#Initialize parameters
W1, b1, W2, b2 = initialize_parameters(n_x, n_h, n_y)

#Iterate and update parameters epoch number of times
for i in range(1,epoch):

    #Forward propagation through two layers
    #Sigmoid activation in the first/hidden layer
    A1, Z1 = forward_prop(W1, b1, X, "Sigmoid")

    #No activation in the second/output layer
    A2, Z2 = forward_prop(W2, b2, A1, "None")

    #Cost function. Mean Squared Error
    cost = cost_func(Y, A2, m)

    #Store cost for later plotting
    cp[i-1] = cost

    #Backward propagation, finding the weights and biases
    dW1, db1, dW2, db2 = backprob(A2, Y, A1, X, W2, m)

    #Update Parameters
    W1 = W1 - learning_rate * dW1
    W2 = W2 - learning_rate * dW2

    b1 = b1 - learning_rate * db1
    b2 = b2 - learning_rate * db2

#Stores the parameters found after training.
parameters = [W1, b1, W2, b2]

# Using matplotlib to illustrate the cost in each iteration.
plt.plot(cp)
plt.title('model cost')
plt.ylabel('Cost')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()

#Print the final and the rooted value of the cost
print("Final Cost: ", cost)
print("Final Cost: ", sqrt(cost))

return parameters
    
```

5.10 RUNNING MODEL 1

With the model finished it is time to run it. If all elements are correctly implemented, the matrix dimensions are correct for all matrices, there are no typos, then the model will, hopefully, run, and one has successfully made an ML model for initial testing. As a check, before running the model, the dimensions of X and Y are printed. As mentioned, for the built system to work, the dimensions of X and Y needs to be

- X : [#features, #examples]
- Y : [#examples, 1]

There may be a need to check the dimensions several times. Because of this, it can be recommended to make a function taking in the four datasets and printing out their dimensions along with descriptive text.

```
#Print the dimensions
def print_shapes(X, Y, Xt, Yt):
    print("X shape: " + str(X.shape))
    print("Y shape: " + str(Y.shape))
    print("Xt shape: " + str(Xt.shape))
    print("Yt shape: " + str(Yt.shape))
```

For the check, some initial values for the hyperparameters are set, X and Y training and test data are loaded, and then the different dimensions are printed.

```
#Set hyperparameters
learning_rate = 0.1
epoch = 3
num_motor=1

#Load X and Y training set and Xt and Yt test set.
X, Y, Xt, Yt = load_treated_data(num_motor)

#Print the dimensions of different datasets
print_shapes(X, Y, Xt, Yt)
X shape: (24, 192)
Y shape: (192, 1)
Xt shape: (24, 31)
Yt shape: (31, 1)
```

From the print, one can see that the dimensions are correct. One can then move on to the final step. For the first test, *epoch* is set to 3, the *learning_rate* is set to a typical initial value; 0.1, and *num_motors* to 1. This is done to see that the model works and that the cost is reduced through each iteration. When running the model again, one may “comment out (#)” the *print_shapes()* function.

```

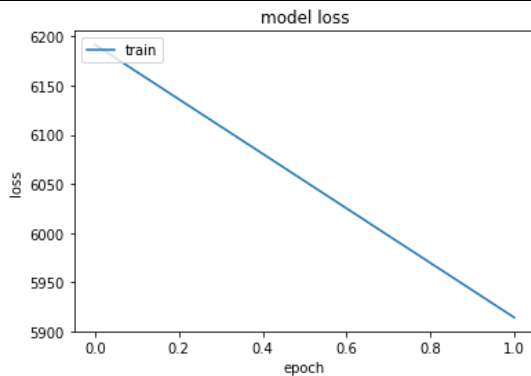
#Set hyperparameters
learning_rate = 0.1
epoch = 3
num_motor=1

#Load X and Y training set and Xt and Yt test set.
X, Y, Xt, Yt = load_treated_data(num_motor)

#Print the dimensions of different datasets
#print_shapes(X, Y, Xt, Yt)
    
```

```

#Run Model 1
parameters = model(X, Y, learning_rate, epoch)
    
```



Final Cost: 5913.893948849516
 Final Squared Cost: 76.90184619922668

The result is illustrated above. The cost can be seen to go down, and the final squared cost is a value that is in the range of what to expect. With a working model, some further testing with different hyperparameters is done to see how the cost changes and can be seen in Figure 29. A thing to note is the value presented from cost squared. It can be used to as an estimate for the average difference between the predicted value and the correct value. Because of the $\frac{1}{2m}$ element in the cost function, its value is not exact, but still a good estimation.

Further tests

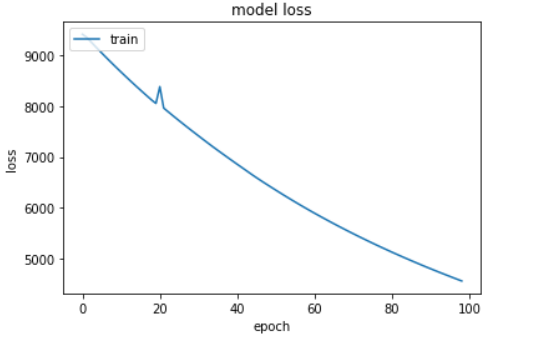
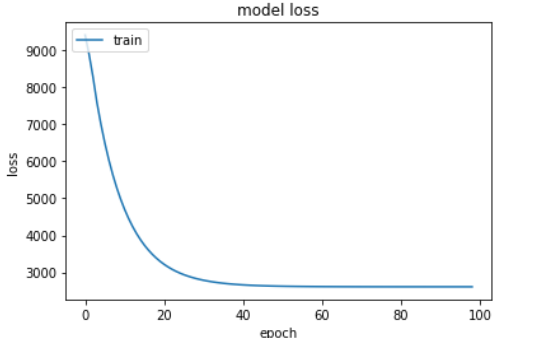
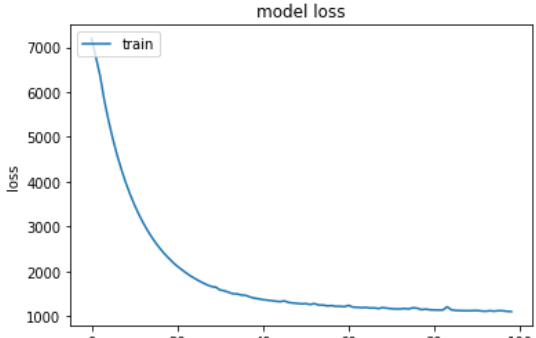
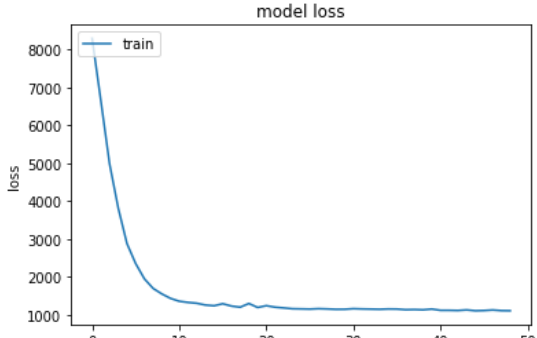
learning_rate = 0.001 epoch = 100 num_motor=5	learning_rate = 0.01 epoch = 100 num_motor=5
 <p>Final Cost: 4558.1684 Final Squared Cost: 67.5142</p>	 <p>Final Cost: 2608.1477 Final Squared Cost: 51.0700</p>
learning_rate = 0.01 epoch = 100 num_motor=50	learning_rate = 0.05 epoch = 50 num_motor=100
 <p>Final Cost: 1100.40837 Final Squared Cost: 33.1724</p>	 <p>Final Cost: 1101.41217 Final Squared Cost: 33.1875</p>

Figure 29: Print of cost of Model 1 trained with different hyperparameters

5.11 KERAS

The Keras model is built to be similar to Model 1, see Figure 18. There are two neurons in the hidden layer and one neuron in the output layer. The first layer has a Sigmoid activation function, and the output layer has none. The weights are initialized randomly, and the biases are initialized as zero. All of the presented code for Keras are implemented into the same file “Keras”.

Libraries used:

```
#Import Keras to build and train ML model
from keras.models import Sequential
from keras.layers import Dense

#Import sklearn for mathematical computation of MSE
from sklearn import metrics

#Import matplotlib for visualization of results
from matplotlib import pyplot as plt

#Import own code for data treatment and to load training and test set
from Data_treatment import *
```

5.11.1 Set hyperparameters and load training and test data:

As in Model 1, *num_motors*, *epochs* and *learning_rate* need to be set, with Keras it is also common to implement *batch_size*, a common optimization algorithm for speeding up training, where the number of training examples are split into training batches, where batch sizes of 2^x are often better [6].

As mentioned, the shape of the input data varies from system to system. With Keras the dimensions of train and test X needs to be [#examples, #features], and train and test Y [#examples, 1]. All of them also need to be of type NumPy array. To test the model the hyperparameters are set to a low value. To load data the function *load_treated_data()* from “Data_treatment” and sub-chapter 5.2.6 is used. To check that the dimensions of X, Y, X_t and Y_t are correct, they are printed using the *print_shapes()* function.


```

#Set hyperparameters
num_motors = 1
batch_size = 1
learning_rate = 0.1
epochs = 15

#Load datasets for training and testing
X, Y, Xt, Yt = load_treated_data(num_motors)

#Print the dimensions of different datasets and number of features
print_shapes(X, Y, Xt, Yt)
    X shape: (24, 192)
    Y shape: (192, 1)
    Xt shape: (24, 31)
    Yt shape: (31, 1)
    
```

Not all dimensions match. To correct this, one can see that X and Xt will match after they are transposed. By printing out “ $X.shape[1]$ ”, one can also check that one has the correct notation for the number of features from input X .

```

#Transposing X(train) and Xt(test)
X = X.T
Xt = Xt.T

#Print the dimensions of different datasets and number of features
print_shapes(X, Y, Xt, Yt)
print("Input shape: ", X.shape[1])
    x shape: (192, 24)
    y shape: (192, 1)
    xt shape: (31, 24)
    yt shape: (31, 1)
    Input shape: 24
    
```

After running the code, all dimensions are now in the desired format.

5.11.2 Building the Keras model

The process of setting up a Keras Model for initial testing can be divided into five steps, Figure 15. In Keras there are several functions that can be added in each step.

5.11.2.1 Select Model Type

The model type *Sequential()* is used and is a good choice for building simple NN. *Sequential()* is model where one stack the layers, or in other words, add one and one layer.

```

#Keras ML model
#input format: [num_ex, num_features]
model = Sequential()
    
```

5.11.2.2 Add layers

In the process of adding layers to the model, one also specifies the characteristics of each layer. First, one set the number of neurons in the layer. For the first layer in Keras one adds the number of features found in X . With *kernel_initializer* and *bias_initializer* one set the initial values for the weights and biases. In the end, the activation function is added.

```
#First hidden layer
model.add(Dense(2, #Neurons in layer
                input_dim=X.shape[1], # Features from input X
                kernel_initializer='random_uniform', #Initialize W
                bias_initializer='zeros', #Initialize b
                activation='sigmoid' #Activation function
            ))

#Output layer
model.add(Dense(1,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'
            ))
```

5.11.2.3 Compile

When all desired layers are added, and the NN is built, one configuration its learning process with *compile*. As with Model 1, mean squared error is chosen as loss function. For the Keras model one has to add an optimizer. There are several optimizers available. For this model, an optimizer is chosen and edited to only take in the learning rate.

```
# Compile model

#Define optimizer: Only learning rate is set.
sgd = optimizers.SGD(lr = learning_rate)

#Compile model with loss function and optimizer
model.compile(loss='mean_squared_error', optimizer='sgd')
```

5.11.2.4 Training the model with model.fit

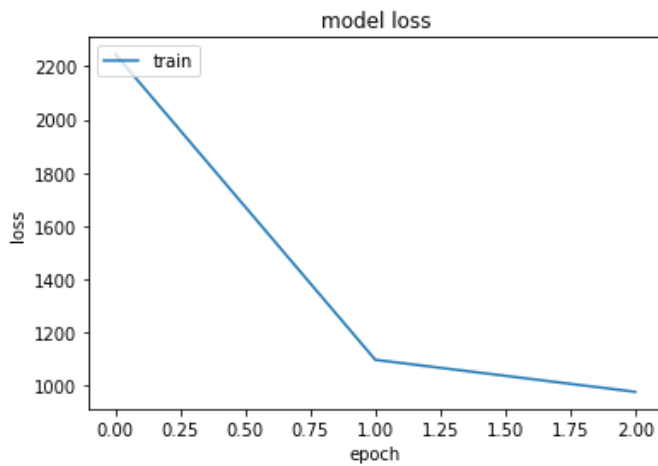
Through “Fit” the model trains the NN. It takes in both X and Y , and as with Model 1, through gradient decent, it trains itself to fit the predicted value from X to the labeled value Y . *Epochs* and *batch_size* are hyperparameters already set. *Verbose* is a Keras function providing the option to visualize the progress of training. Verbosity mode 0 = silent, 1 = progress bar and 2 = one line per epoch. Setting history equal to *model.fit()* saves the results to history.

```
# Fit/ train the model
history = model.fit(X, Y, epochs= epochs, batch_size = batch_size,
                   verbose=0)
```

Printing out the lowest loss from history give the lowest MSE found through training.

```
#Plot cost/ loss in each epoch
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

print('Train lowest mse: ' + str(min(history.history['loss'])))
print('Train lowest sqrt(mse): ' +
      str(np.sqrt(min(history.history['loss']))))
```



```
Train lowest mse: 977.9868474650478
Train lowest sqrt(mse): 31.272781255671006
```

Further tests

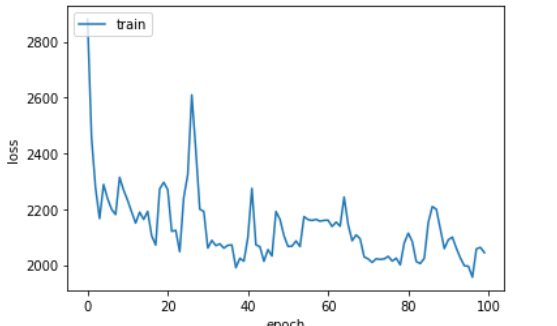
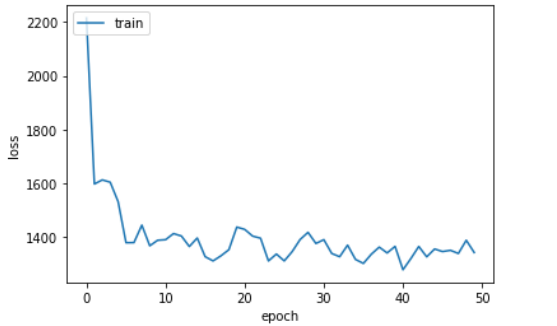
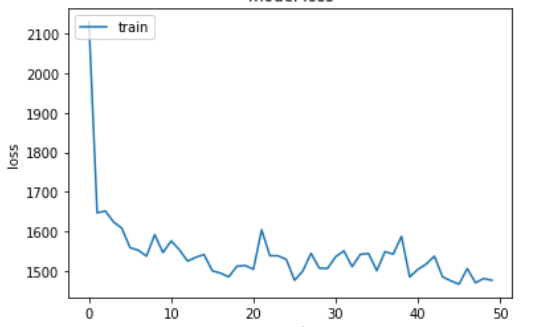
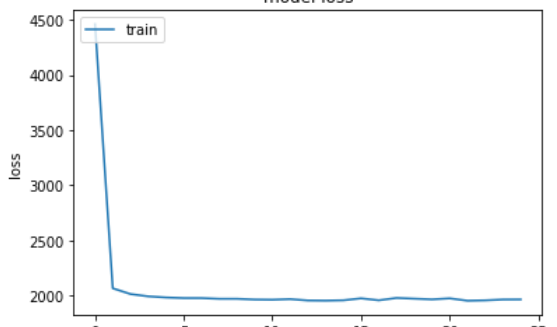
<pre>num_motors = 5 batch_size = 1 learning_rate = 0.01 epochs = 100</pre>	<pre>num_motors = 10 batch_size = 4 learning_rate = 0.05 epochs = 50</pre>
<p style="text-align: center;">model loss</p>  <p>Train lowest mse: 1956.2362 Train lowest sqrt(mse): 44.22936</p>	<p style="text-align: center;">model loss</p>  <p>Train lowest mse: 1277.85963 Train lowest sqrt(mse): 35.74716</p>
<pre>num_motors = 25 batch_size = 8 learning_rate = 0.08 epochs = 50</pre>	<pre>num_motors = 100 batch_size = 256 learning_rate = 0.01 epochs = 25</pre>
<p style="text-align: center;">model loss</p>  <p>Train lowest mse: 1467.785452 Train lowest sqrt(mse): 38.3116</p>	<p style="text-align: center;">model loss</p>  <p>Train lowest mse: 1952.45685 Train lowest sqrt(mse): 44.186</p>

Figure 30: Print of cost from Keras model trained with different hyperparameters

5.11.2.5 Evaluate

If one manages to make a NN that sufficiently good at predicting, the next step is to evaluate how good the model is at predicting the RUL value for an independent test set. Through the function *predict()*, Keras use the model built to find predicted values on the test dataset X_t . Through sklearn the MSE is calculated over all examples in X_t and Y_t . To find the rooted score, the value found is squared.

```
#predict:
pred = model.predict(Xt)

# evaluate the model
score = metrics.mean_squared_error(pred, Yt)
print("Test score: ", score)

score_root = np.sqrt(score)
print("Rooted test score: ", score_root)
```

```
Score: 4336.704481331986
Rooted score: 65.85365958951701
```


6 DISCUSSION

6.1 SOURCES

Web pages

In this thesis, several of the sources used are web pages, web-articles and web guides. Many of them, written by authors with Ph.D. in programming, NN, ML or deep learning. These sources are used to get an idea about the usefulness and popularity of PdM, ML and TL. They are used to understand both common and specific topics and as guidance for building the two ML models.

By using webpages, there can be said to be a degree of risk for a biased author or an author presenting misunderstood or false information. To mitigate this, each web-article are mainly used as one of several to show a common opinion. In other cases, the publisher or the background of the writer is evaluated.

Several of the articles and guides found are also written in an informal style. On the positive side, this makes the description and explanation of difficult topics more figurative and easier to understand. Through his course, Ng repeatedly states that it is often more important to intuitively understand what is happening in an NN and to exactly know everything that happens. A negative side is that the authors own understanding and opinions are often expressed and the information shared is less generalized. This results in topics being explained in several different ways. It can thus be hard to find the common thread or to choose one explanation. Still, a benefit found is the presentation of personal experiences, problems and solutions, which are applicable to one's own problem.

Coursera Online Course by Andrew Ng [6]

One of the main sources used in this thesis is the online course “Neural Networks and Deep Learning” at Coursera, mainly presented and made by Andrew Ng. Even though only one source, it can be argued to be a reliable source. Andrew Ng has a long experience working with ML, NN and deep learning. As mentioned, he is credited as one of the main drivers behind deep NN. He has worked as head of Baidu AI and Google brain and has given several presentations about ML, his Coursera course being one of the latest ones.

Still, as Ng[6] is one source, to verify, similar and corresponding information can be found in the book by Coelho and Richert [5], by Nielsen [8], and in the many web articles used. One of the main differences experienced, also the main reason for choosing Ng as a main source. He is a recommended source for the overall presentation of ML and to give intuitive information enabling one to easier understand the topic.

6.2 THE POTENTIAL AND INTEREST OF PdM, ML AND TL

As described in this thesis and referenced in several articles and web articles, there is an arguably potential and interest for PdM, ML and TL. With the aim of PdM of predicting faults and failures before it happens, there is a possibility to greatly reduce, or even remove, the case of unplanned downtime, or failure. This aim is in and of itself an argument, as unplanned downtime can be extremely costly. Another question then becomes if the aim of PdM is realistic. The report by IoT Analytics can be said to answer this. Through studying several maintenance companies from several countries, they have found many believing in its potential. This can also be seen by their prediction of an annual growth rate of 40% for PdM and annual technology spending reaching of almost 11 Billion by 2022. Without arguing about the correctness of the numbers, the report still presents a considerable interest. Further interest is found in the study by Mulders and Haarman [15], where few already employ people and tools needed, but many are ambitious about its potential.

Several authors argue ML to be highly relevant for PdM. An aim of ML is to predict or decide based input data. An aim of PdM to predict potential faults from sensor data. These two aims can be said to be similar and show that ML can be an enabler of PdM. As computational power increases, the ML tools are improved, and companies are digitalized, there is a rising potential and market for ML. For PdM one of the most common arguments through the articles read is that ML is crucial to make use of data collected and thus enabling PdM.

A challenge presented by Karsten Moholt is that there is much data collected from running machines, but comparably little data connected from faults, thus little training data. A solution could be to share the data, but as the much of the data is sensitive to one company, it is not easily shared. A solution may be TL. By being able to transfer data from one motor to another, one can predict failure without having registered failure data for that exact machine, thus enabling one to train an NN with less available data. TL can also be said to bring with it a possibility where companies are willing to shear their data to build and train ML models that in return can be useful to them.

Through the theory chapter 2.1.3, there is an arguably need for research on ML for PdM. One of the first arguments are the findings from the study by Mulders and Haarman (2017). They have found many interested in implementing PdM, but few having reached a level of PdM. They state few reference cases as one of the main challenges. IEEE has also sent out a call for papers regarding ML and PdM, further arguing the case of PdM and thus ML for PdM needing research.

6.3 UNDERSTANDING MACHINE LEARNING

This thesis aims to be an introduction and to provide a practical start-guide for ML. To achieve this, this thesis give an introduction to what ML is, how it works, and how one can build an ML model to start with initial testing. These questions are answered in four steps, firstly in the theory chapter and the in the following three chapters. This setup is done to answer the questions and to present the topic by starting wide and to stepwise going into the details, down to code and practical examples. It is also done to make the learning process an iterative process.

ML itself can be argued to be a stepwise and iterative process. Also mentioned by Ng [6] and Coelho and Pedro [5], ML is not only iterative when training itself, its use and the process of achieving good results is also an iterative process. One has to test the different hyperparameters and see which ones give the best result. The process of learning and understanding ML can also be argued to be iterative. Thus, the process of presentation of information is done through iterative steps.

Another choice could have been to present the topic thoroughly, from start to finish, by presenting all the elements along with general code. This could have been a better solution as it could have given an easier way to thoroughly understand each sub-topic or element of ML. Instead of explaining an element, then another, and only to in the next chapter explain it again. The stepwise and iterative solution was still chosen. There were several reasons for this. One of them is that there are many elements to introduce and to consider when understanding and building an ML model. When presenting all of them one after another, the information may be confusing as it becomes hard to see how it is all connected or to understand how one is to practically implement this information.

When starting to present the topic of ML, several of its elements may seem like black art. As also experienced by Coelho and Pedro [5]. Starting by presenting code or by using the NN libraries as Keras or Tensorflow, one might get a system working but may not understand why it is not giving good results. By not understanding how ML works, it may also be hard to understand other articles presenting ideas and solutions for how to improve the network. On the other hand, by only reading theory, there may be several concepts that are hard to understand without practical experience. To learn and understand ML it is thus recommended to first get some intuitions about what ML and how it works. Even if some elements are unclear, it is recommended to move on and get some practical experience by programming and see how the learning and training process of the NN works. Then one can go back and read the theory again. Hopefully, the theory is now more understandable, and one can learn how to make an improved model.

To understand ML and to know the answers to either one of the presented questions, one is likely to need some understanding about all three questions. It may be hard to understand what ML is without knowing how it works. Moreover, it may be hard to know how it works without knowing how to build. Thus, only by understanding all three questions may one understand ML.

6.4 BUILDING AND PROGRAMMING AN ML MODEL

When building an ML model, there are many different choices to make regarding which tools to use, how to handle data and how to build and design the NN. First, there are several choices for which programming language to use. Then one can choose a framework like Keras or Tensorflow if one is to use a framework at all. Thirdly, libraries for visualization, data treatment, number operations and matrix operation, to mention some, must also be implemented. Further, there are many choices to make regarding the ML system itself.

To download python, needed environments, libraries and frameworks the Anaconda Navigator was used. The reason for this was that it enables all ML relevant tools to be collected at one place. It contains applications as python and python notebooks, environments, libraries and frameworks, along with the option to download missing libraries. An alternative could have been to download the needed libraries through command. If one is used to work with command, this approach may be quicker to work with and does not require one to download a navigator.

For python programming, the web-based notebook Jupyter is used. It has some shortcomings; as it is run from the browser, it is challenging to structure longer code and the file-format is “.ipynb” instead of the standard python format, “.py”. One effect that can be experienced when implementing code and functions from other files, as done with the “*Data_treatment*”. Still, for initial testing and exploration Jupyter is recommended. As the code is structured in a cell format and each cell can be run independently, it is used to continuously test different elements of the code, and thus build one and one element.

To program a basic NN oneself has both benefits and drawbacks and can be said to depend on the goal of building an NN. One of the main benefits to mention is that it gives an increased understanding of how the NN works, and how one can improve the system. The basics of an NN can be said to be equal for almost all other NN and ML models. A negative side is that it takes time to learn and get a sufficient understanding of all elements in an NN. There is also a certain need to know how python works, how to treat matrices, and generally to know calculus. As an example, Nielsen mentions in his book that finding the correct formulas for backpropagation is both difficult and frustrating. Thus, using premade libraries as Tensorflow and Keras may be a better choice for making good ML models.

Through Keras’s webpage, they advertise Keras as an NN API for easy and fast prototyping. As shown through the programming example, this can be argued to be true. With few lines of code, one can start with initial testing. For improved models, one can easily add several more layers or change the number of neurons in a layer. A major benefit with Keras, also found in Tensorflow, is that they have backpropagation built into their libraries. With this, one does not have to mathematically find the partial derivatives of the cost function, which becomes increasingly complex as one increases the number of layers. A popular alternative to Keras is Tensorflow. It is the most popular deep learning library. Compared to Keras, Tensorflow offers more freedom in designing and building ML models and offers more advanced operations. With Tensorflow one also has a higher degree of insight into what is happening in the NN. One aim of this thesis is to provide a guide for initial testing. Thus Keras was chosen as it offers easy and fast prototyping and is a good tool to start building ML models.

6.5 USEFULNESS

As an introduction and start-guide this thesis main usefulness is, hopefully, towards the intended readers, to help with a better understanding about ML, and as a guide towards building ML models. By replicating the code presented in this guide, one has a basis model that one can build on, and, hopefully, an understanding about NN and ML. With this, one may have a better understanding towards implemented solutions and ideas presented by others.

A limiting factor is that the thesis presents ML up to the point of running the first test on the training data. There are thus many elements of ML that are not presented in this thesis that are important in the process to further develop the ML model. Another limiting factor is that the thesis is not tested and evaluated by the intended reader group or validated by experts in the field of ML. This could have enabled the guide and code to have better credibility and to work better as an introduction and start guide.

Still, as a start guide for ML, this thesis can be argued to stand out. As previously mentioned, and used in this thesis, there are several other books, guides, courses and webpages presenting the topic of ML. Working with these guides, most are found to focus on classification problems. The general case is also that a guide either presents the code or the theory. Thus, to be a start guide for those interested, but without excessive knowledge of programming, this thesis presents both elements. None of the other guides are found to focus on presenting the topic and to be a practical start guide for ML for PdM. As there is a rising interest and need for research on the combination of these fields, there is arguably also a need to make guides to these specific cases. This is also the focus of this thesis, and arguably where it stands out.

Thus, this thesis can be used in different cases and settings where there is an interest in learning about ML and its practical implementation. One scenario may be for classes taught to maintenance engineers or others, where it can be used as an exercise document. Another is to help other master students and researchers relatively quickly get started on building ML models, to develop them further and to use ML in case studies. A third scenario is for the maintenance company Karsten Moholt, or other companies, where it can be used to increase the knowledge about ML or enable them to get started with ML.

7 CONCLUSION

In this thesis Predictive Maintenance (PdM), Machine Learning (ML) and Transfer Learning (TL) are presented and answers are provided to the questions; what is Machine Learning, how does it work, and how can one build a Machine Learning model to start with initial testing. Through is an introduction and a practical start guide for those interested in using ML for PdM, is provided.

PdM is presented as a technique for monitoring operating condition to provide data that can ensure the maximum interval between repairs and minimize the number and cost of unscheduled machine failures. ML is found as an important and powerful tool for finding patterns and make predictions from a vast amount of data, where Neural Networks (NN) is the main technique for implementing ML. TL is found as a powerful idea, where knowledge from one NN can be transferred to another. All these topics are found to have considerable future potential and are in need of development and research.

The second part of this thesis provides answers to the given questions and, hopefully, enables an understanding towards ML. Theory from several sources are collected and presented providing a guide for information and background about ML and NN. To enable the reader to start programming, tools such as the programming language Python, the notebook Jupyter, the mathematical library NumPy, data structure library Pandas, and the high-level NN API, Keras are presented, used and recommended. Two ML models are programmed for a Remaining Useful Life (RUL) regression problem on the Turbofan Degradation Dataset by NASA. The first model provides a guide on how to program an NN oneself, thus getting a better understanding toward ML, and includes the elements; “prepare data”, “initialize weights and biases”, “forward propagation”, “calculating cost”, “backpropagation” and “update parameters”. The second model presents how to program the same model with Keras, enabling one to program ML models with few lines of code.

The guide enables a foundation for further development of ML models for different PdM scenarios. The thesis provides arguments and justification for the need of further development of ML solutions and research connected to PdM. It is recommended to work iteratively with the topic of ML, to program a basic ML model to understand the process and system and to continue testing and prototyping with Keras.

8 FURTHER WORK

As an introduction and start-guide for ML in PdM this thesis is the first iteration. For further work, several further iterations are recommended. An important step, not done in this thesis, is to present the document to experts in ML and the intended target group. Through feedback and the code and information can be further developed and improve the thesis as a guide. This enables the guide and code to have better credibility and to work better as an introduction and start guide.

Another case is to further develop the guide by adding information. One example is to add a guide on the popular NN platform TensorFlow. A second example could be to provide a guide on how to build more advanced NN for PdM. There are several implementations, functions and solutions available that can improve the NN. By implementing this, the guide may not only work as an introduction to get started, but also as a document to presenting how to build finished ML models for PdM.

REFERENCES:

1. Fogoros, L. *Connected sensors & machine learning – two current trends in predictive maintenance*. [cited 2018 09.03]; Available from: <http://iiot-world.com/predictive-maintenance/connected-sensors-ml-two-current-trends-in-predictive-maintenance/>.
2. Cline, B., et al. *Predictive maintenance applications for machine learning*. in *2017 Annual Reliability and Maintainability Symposium (RAMS)*. 2017.
3. Li, H., et al., *Improving rail network velocity: A machine learning approach to predictive maintenance*. *Transportation Research Part C: Emerging Technologies*, 2014. **45**: p. 17-26.
4. Moholt, K. [cited 2017 18.12.17]; Available from: <http://karstenmoholt.com/About>.
5. Coelho, L.P. and W. Richert, *Building machine learning systems with Python*. 2015: Packt Publishing Ltd.
6. A. Ng, K.K., Y. B. Mourri, *Neural Networks and Deep Learning* 2017, Coursera: www.coursera.org.
7. Brownlee, J. *Machine Learning Mastery*. [cited 2018 22.06]; Available from: <https://machinelearningmastery.com>.
8. Nielsen, M.A., *Neural Networks and Deep Learning* ed. D. Press. 2015.
9. Chiu, Y.-C., F.-T. Cheng, and H.-C. Huang, *Developing a factory-wide intelligent predictive maintenance system based on Industry 4.0*. *Journal of the Chinese Institute of Engineers*, 2017. **40**(7): p. 562-571.
10. Amruthnath, N. and T. Gupta, *A Research Study on Unsupervised Machine Learning Algorithms for Fault Detection in Predictive Maintenance*. 2018.
11. Mobley, R.K., *An introduction to predictive maintenance*. 2002: Butterworth-Heinemann.
12. Sciban, R. *6 Tools for a Successful Predictive Maintenance Program*. 2017 [cited 2018 06.06]; Available from: <https://us.hitachi-solutions.com/blog/6-tools-for-a-successful-predictive-maintenance-program/>.
13. Scully, P. *New Report Indicates US\$11 Billion Predictive Maintenance Market By 2022, Driven By IoT Technology And New Services*. 2017 [cited 2018 06.06]; Available from: <https://iiot-analytics.com/report-us11-billion-predictive-maintenance-market-by-2022/>.
14. 2018, I.I. *Special Sessions Call for Papers*. 2016 [cited 2018 15.05]; Available from: <https://icnsc2018.jnu.edu.cn/MLPAIA.html>.
15. M. Mulders, M.H. *Predictive Maintenance 4.0 - Predict the unpredictable*. 2017.
16. Nowitz, A. *The Economics of the Smart Factory How does Machine Learning Lower the Cost of Asset Maintenance Part 1*. 2017 [cited 2018 05.06]; Available from: <http://www.presenso.com/single-post/2017/05/24/the-economics-of-the-smart-factory-how-does-machine-learning-lower-the-cost-of-asset-maintenance-part-1/>.
17. Kelly, K. *The Three Breakthroughs That Have Finally Unleashed AI on the World*. 2014 [cited 2018 06.06]; Available from: <https://www.wired.com/2014/10/future-of-artificial-intelligence/>.
18. Andre, E., et al., *Dermatologist-level classification of skin cancer with deep neural networks*. *Nature*, 2017. **542**(7639): p. 115.
19. LeCun, Y., Y. Bengio, and G. Hinton, *Deep learning*. *Nature*, 2015. **521**: p. 436.
20. Øverlier, L. and A. Aspelund, *Intellectual Property and Machine Learning: An exploratory study*. 2017.
21. Marr, B. *27 Incredible Examples Of AI And Machine Learning In Practice*. 2018 [cited 2018 20.06]; Available from: <https://forbes.com/sites/bernardmarr/2018/04/30/27-incredible-examples-of-ai-and-machine-learning-in-practice/#49d1faaf7502>.

22. Irwin, R. *Predictive Maintenance and Machine Learning Revolutionizing Reliability*. [cited 2018 05.06]; Available from: <https://reliabilityweb.com/articles/entry/predictive-maintenance-and-machine-learning-revolutionizing-reliability>.
23. MSV, J. *How Machine Learning Enhances The Value Of Industrial Internet of Things*. 2017 [cited 2018 05.06]; Available from: <https://www.forbes.com/sites/janakirammsv/2017/08/27/how-machine-learning-enhances-the-value-of-industrial-internet-of-things/#6ed43b163f38>.
24. Ovenden, J. *Machine Learning Top Trends In 2017*. 2017 [cited 2018 05.06.18]; Available from: <https://channels.theinnovationenterprise.com/articles/machine-learning-top-trends-in-2017>.
25. Marr, B. *What Is The Difference Between Deep Learning, Machine Learning and AI?* 2016 [cited 2018 06.06]; Available from: <https://www.forbes.com/sites/bernardmarr/2016/12/08/what-is-the-difference-between-deep-learning-machine-learning-and-ai/#7aa2d60e26cf>.
26. Buczkowski, A. *What's the difference between Artificial Intelligence, Machine Learning and Deep Learning?* 2018 [cited 2018 06.06]; Available from: <http://geoawesomeness.com/whats-difference-artificial-intelligence-machine-learning-deep-learning/>.
27. Reese, H. *Understanding the differences between AI, machine learning, and deep learning*. 2017 [cited 2018 06.06]; Available from: <https://www.techrepublic.com/article/understanding-the-differences-between-ai-machine-learning-and-deep-learning/>.
28. Copeland, M. *What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?* 2016 [cited 2018 06.06]; Available from: <https://blogs.nvidia.com/blog/2016/07/29/whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/>.
29. Parrish, K. *Deep learning vs. machine learning: what's the difference between the two?* 2018 [cited 2018 06.06]; Available from: <https://www.digitaltrends.com/cool-tech/deep-learning-vs-machine-learning-explained/2/>.
30. Liu, Q. and Y. Wu, *Supervised Learning*, in *Encyclopedia of the Sciences of Learning*, N.M. Seel, Editor. 2012, Springer US: Boston, MA. p. 3243-3245.
31. *Unsupervised Learning*, in *Encyclopedic Reference of Genomics and Proteomics in Molecular Medicine*. 2006, Springer Berlin Heidelberg: Berlin, Heidelberg. p. 1972-1972.
32. Brownlee, J. *Supervised and Unsupervised Machine Learning Algorithms*. 2016.
33. Izquierdo, L.R. and S.S. Izquierdo, *Reinforcement Learning*, in *Encyclopedia of the Sciences of Learning*, N.M. Seel, Editor. 2012, Springer US: Boston, MA. p. 2796-2799.
34. Arulkumaran, K., et al., *Deep Reinforcement Learning: A Brief Survey*. IEEE Signal Processing Magazine, 2017. **34**(6): p. 26-38.
35. *Semi-supervised Learning*, in *Encyclopedia of the Sciences of Learning*, N.M. Seel, Editor. 2012, Springer US: Boston, MA. p. 3036-3036.
36. Susto, G.A., et al., *An adaptive machine learning decision system for flexible predictive maintenance*. Vol. 2014. 2014. 806-811.
37. A. Ng, K.K., Y. B. Mourri, *Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization*. 2017, Coursera: www.coursera.org.
38. *Transfer Learning*, in *Encyclopedia of the Sciences of Learning*, N.M. Seel, Editor. 2012, Springer US: Boston, MA. p. 3337-3337.
39. Weiss, K., T.M. Khoshgoftaar, and D. Wang, *A survey of transfer learning*. Journal of Big Data, 2016. **3**(1): p. 9.
40. Pan, S.J. and Q. Yang, *A Survey on Transfer Learning*. IEEE Transactions on Knowledge and Data Engineering, 2010. **22**(10): p. 1345-1359.
41. Ruder, S. *Transfer Learning - Machine Learning's Next Frontier*. 2017 [cited 2018 12.06]; Available from: <http://ruder.io/transfer-learning/>.

42. Jillian, S. *How is Mtell Able to Transfer Learned Behaviors from One Machine to Another, While Some Claim all Machines are Unique?* 2013 [cited 2018 15.06]; Available from: <http://mtell.com/transfer-learning-for-machines>.
43. Tunkelang, D. *10 Things Everyone Should Know About Machine Learning*. 2017 [cited 2018 20.06]; Available from: <https://medium.com/@dtunkelang/10-things-everyone-should-know-about-machine-learning-15279c27ce96>.
44. Brownlee, J. *Discover Feature Engineering, How to Engineer Features and How to Get Good at It*. 2014 [cited 2018 20.06]; Available from: <https://machinelearningmastery.com/discover-feature-engineering-how-to-engineer-features-and-how-to-get-good-at-it/>.
45. learn, s. 4.3. *Preprocessing data*. [cited 2018 21.06]; Available from: <http://scikit-learn.org/stable/modules/preprocessing.html>.
46. learn, s. *Importance of Feature Scaling*. [cited 2018 20.06]; Available from: http://scikit-learn.org/stable/auto_examples/preprocessing/plot_scaling_importance.html.
47. Li, J., et al., *Feature Selection: A Data Perspective*. Vol. 50. 2016.
48. Jupyter. *What is the Jupyter Notebook?* [cited 2018 22.06]; Available from: http://jupyter-notebook-beginner-guide.readthedocs.io/en/latest/what_is_jupyter.html.
49. Oslo, U.i. *Jupyter Notebook og Jupyterhub*. 2017 [cited 2018 22.06]; Available from: <https://uio.no/studier/emner/matnat/ibv/BIOS1100/h17/ressurser/jupyterhub.md>.
50. Anaconda. *What is Anaconda*. [cited 2018 20.06]; Available from: <https://www.anaconda.com/what-is-anaconda/>.
51. Anaconda. *Anaconda Navigator*. [cited 2018 20.06]; Available from: <https://anaconda.org/anaconda/anaconda-navigator>.
52. pandas. *pandas 0.23.0 documentation*. [cited 2018 11.06]; Available from: <http://pandas.pydata.org/pandas-docs/stable/index.html>.
53. pandas. *The pandas project*. [cited 2018 11.06]; Available from: <https://pandas.pydata.org/about.html>.
54. Scikit-learn. *Home*. [cited 2018 22.06]; Available from: <http://scikit-learn.org/stable/index.html#>.
55. Keras. *Keras: The Python Deep Learning library*. [cited 2018 11.06]; Available from: <https://keras.io/>.
56. TensorFlow. *About TensorFlow*. 2018 [cited 2018 10.04.18]; Available from: <https://www.tensorflow.org/>.
57. H2O. *About Us*. [cited 2018 11.06]; Available from: <https://www.h2o.ai/about/>.
58. NASA. *PCoE Datasets*. [cited 2018 11.06]; Available from: <https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/>.
59. NASA. *NASA's Open Data Portal*. [cited 2018 05.06.18]; Available from: <https://data.nasa.gov/stories/s/e3dt-gtde>.
60. kaggle. *Get data*. [cited 2018 05.06]; Available from: <https://www.kaggle.com/about/datasets/get>.
61. Saxena, A., et al. *Damage propagation modeling for aircraft engine run-to-failure simulation*. in *Prognostics and Health Management, 2008. PHM 2008. International Conference on*. 2008. IEEE.

APPENDIX A: TURBOFAN ENGINE DEGRADATION SIMULATION DATA SET, README – FROM ZIPFILE DOWNLOADED FROM NASA.

Data Set: FD001

Train trajectories: 100

Test trajectories: 100

Conditions: ONE (Sea Level)

Fault Modes: ONE (HPC Degradation)

Data Set: FD002

Train trajectories: 260

Test trajectories: 259

Conditions: SIX

Fault Modes: ONE (HPC Degradation)

Data Set: FD003

Train trajectories: 100

Test trajectories: 100

Conditions: ONE (Sea Level)

Fault Modes: TWO (HPC Degradation, Fan Degradation)

Data Set: FD004

Train trajectories: 248

Test trajectories: 249

Conditions: SIX

Fault Modes: TWO (HPC Degradation, Fan Degradation)

Experimental Scenario

Data sets consists of multiple multivariate time series. Each data set is further divided into training and test subsets. Each time series is from a different engine – i.e., the data can be considered to be from a fleet of engines of the same type. Each engine starts with different degrees of initial wear and manufacturing variation which is unknown to the user. This wear and variation is considered normal, i.e., it is not considered a fault condition. There are three operational settings that have a substantial effect on engine performance. These settings are also included in the data. The data is contaminated with sensor noise.

The engine is operating normally at the start of each time series, and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure. In the test set, the time series ends some time prior to system failure. The objective of the competition is to predict the number of remaining operational cycles before failure in the test set, i.e., the number of operational cycles after the last cycle that the engine will continue to operate. Also provided a vector of true Remaining Useful Life (RUL) values for the test data.

The data are provided as a zip-compressed text file with 26 columns of numbers, separated by spaces. Each row is a snapshot of data taken during a single operational cycle, each column is a different variable. The columns correspond to:

- 1) unit number
- 2) time, in cycles
- 3) operational setting 1
- 4) operational setting 2
- 5) operational setting 3
- 6) sensor measurement 1
- 7) sensor measurement 2
- ...
- 26) sensor measurement 26

Reference: A. Saxena, K. Goebel, D. Simon, and N. Eklund, “Damage Propagation Modeling for Aircraft Engine Run-to-Failure Simulation”, in the Proceedings of the 1st International Conference on Prognostics and Health Management (PHM08), Denver CO, Oct 2008.

APPENDIX B: BACKPROPAGATION - MATHEMATICAL CALCULATION OF dW AND db

A^2 is found through two steps of forward propagation in Model 1.

$$A^{[2]} = \sigma(W^{[2]} * \sigma(W^{[1]} * X + b^{[1]}) + b^{[2]}),$$

$\sigma = \text{activation function}$

Mean Squared Error cost function:

$$C = Cost = \frac{1}{2m} * \sum_{i=1}^m (A^{[2]} - Y)^2$$

Inserting A^2 into the cost function

$$C = \frac{1}{2m} * \sum_{i=1}^m (\sigma(W^{[2]} * \sigma(W^{[1]} * X + b^{[1]}) + b^{[2]}) - Y)^2$$

Using partial derivatives

$$dW^2 = \frac{dC}{dW^{[2]}} = \frac{dC}{dA^{[2]}} * \frac{dA^{[2]}}{dZ^{[2]}} * \frac{dZ^{[2]}}{dW^{[2]}}$$

$$\frac{dC}{dA^{[2]}} = \frac{2}{2m} \sum (A^{[2]} - Y)$$

$$\frac{dA^{[2]}}{dZ^{[2]}} = \text{Derivative of Activation function}$$

There is no activation function in second/ output layer give

$$\frac{dA^{[2]}}{dZ^{[2]}} = 1, \quad A^{[2]} = Z^{[2]}$$

$$\frac{dZ^{[2]}}{dW^{[2]}} = A^1$$

Finding dW2

$$\frac{dC}{dW^{[2]}} = \frac{1}{m} \sum ((A2 - Y) * 1 * A1)$$

$$\frac{dC}{dW^{[2]}} = \frac{1}{m} ((A2 - Y) \circ A1), \text{ der } \sum(a * b) = a \circ b$$

Finding db2

$$db2 = \frac{dC}{db2} = \frac{dC}{dA^{[2]}} * \frac{dA^{[2]}}{dZ^{[2]}} * \frac{dZ^{[2]}}{db^{[2]}}$$

$$\frac{dZ2}{db2} = 1$$

$$\frac{dC}{db2} = \frac{1}{m} \sum ((A2 - Y) * 1 * 1)$$

Finding dW1

$$dW1 = \frac{dC}{dW1} = \frac{dC}{dA^2} * \frac{dA^2}{dZ^2} * \frac{dZ^2}{dA^1} * \frac{dA^1}{dZ1} * \frac{dZ1}{dW1}$$

$$\frac{dZ2}{dA1} = W2$$

$$\frac{dA1}{dZ1} = \text{Derivative of Activation function}$$

$$\frac{dA1}{dZ1} = A1 * (1 - A1)$$

$$\frac{dZ1}{dW1} = X$$

$$dW1 = \frac{1}{m} \sum ((A2 - Y) * 1 * W2 * A1 * (1 - A1) * X)$$

Finding db1

$$db1 = \frac{dC}{db1} = \frac{dC}{dA^2} * \frac{dA^2}{dZ^2} * \frac{dZ^2}{dA^1} * \frac{dA^1}{dZ^1} * \frac{dZ^1}{db1}$$

$$\frac{dZ^1}{db1} = 1$$

$$db1 = \frac{1}{m} \sum ((A2 - Y) * 1 * W2) * A1 * (1 - A1)$$

APPENDIX C - CODE

Code from “Split_data.ipynb” and “Split_data.py”

```

#Import Pandas to load and for data
import pandas as pd
#Import NumPy for array treatment and matrix multiplication
import numpy as np
#Import os for file- and folder-treatment
import os

#Function for removing unnamed columns
def remove_unnamed(df):
    return df.loc[:, ~df.columns.str.contains('^Unnamed')]

#Function for splitting txt-file into several csv-files containing a mot
or each
def split_data(file_folder):
    data_type = file_folder[0].split('_')[0]
    column1 = ['unit', 'cycle', 'setting1', 'setting2', 'setting3']
    column2 = ['sensor{}'.format(i) for i in range(1,22)]
    cols = column1+column2
    os.makedirs('{}_data'.format(data_type))
    for file in file_folder:
        folder = file.split('.')[0]
        print("creating folder data/{}".format(folder))
        os.makedirs('{}_data/{}'.format(data_type, folder))

    # process files
    for j, file_name in enumerate(file_folder):
        path = "{}".format(file_name)
        data = open(path).readlines()
        my_new_data = list(map(lambda line: line.strip('\n').split(' ')[
:-2], data))
        for k in list(range(1,101)):
            new_array = list(filter(lambda x: x[0]=='{}'.format(k), my_n
ew_data))
            df = pd.DataFrame(new_array, columns=cols)
            print("processing file {} folder {}_FD00{} motor{} data".fo
rmat(file_name, data_type, j+1, k))
            df.to_csv('{}_data/{}_FD00{/motor{}.csv'.format(data_type,
data_type, j+1, k))

#Array containing the string names of train and test txt files
training_files = ['train_FD00{}.txt'.format(k) for k in range(1,5)]
test_files = ['test_FD00{}.txt'.format(k) for k in range(1,5)]

#Run through test and training txt-files to split data
for file_folder in [training_files, test_files]:
    split_data(file_folder)
    
```

Code from “Data_treatment.ipynb” and “Data_treatment.py”

```
#Import NumPy for array treatment and matrix multiplication
import numpy as np

#from sklearn import linear_model
import sklearn
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler

#Import Pandas to load and for data
import pandas as pd
```

```
#Function to remove the unnamed column added to dataset through the code for splitting the original dataset.
def remove_unnamed(df):
    return df.loc[:, ~df.columns.str.contains('^Unnamed')]
```

```
#Loads data from path. Return Matrix X with shape [#examples, #features] and a list Y.
def load_data(path):

    #Reads the CSV file and removes unnamed columns
    data = remove_unnamed(pd.read_csv(path)) #Reads the CSV file and removes unnamed columns

    Y = list(list(data['cycle'])) #Makes a list containing the cycle numbers. From 1 to n cycles before failure
    Y.reverse() #Reverse the list to make Y corresponding to RUL for each example in X.

    #Retrieve a list containing values from columns setting1 to sensor21

    X = data.loc[:, 'setting1':'sensor21'].as_matrix()
    return X, Y
```

```
#Load n number of motors. Return array X and Y with dimensions [num_ex, 24], [num_ex, ]
def load_n_motors(num_motor):

    #Create two arrays containing string names for train and test FD001
    training_folders = ['train_FD00{}'.format(i) for i in range(1,5)]
    test_folders = ['test_FD00{}'.format(i) for i in range(1,5)]

    # Folder number. Here: train_FD001 or test_FD00
    k = 0

    #Make an empty Y train and test list
    Y_train = []
    Y_test = []
```

```
#Load num motor X and Y training- and test-sets
for i in range (1, num_motor+1):

    #Create training and test path
    train_path = 'train_data/{}/motor{}.csv'.format(training_folders
[k],i)
    test_path = 'test_data/{}/motor{}.csv'.format(test_folders[k],i)

    #Load temporary X and Y sets
    X_train_temp, Y_train_temp = load_data(train_path)
    X_test_temp, Y_test_temp = load_data(test_path)

    #Build Y train and test list
    Y_train = Y_train + Y_train_temp
    Y_test = Y_test + Y_test_temp

    #Convert Matrix X to a pandas dataframe
    X_train_temp_df = pd.DataFrame(X_train_temp)
    X_test_temp_df = pd.DataFrame(X_test_temp)

    #Initialize first motor
    if (i == 1):
        X_train_df = X_train_temp_df
        X_test_df = X_test_temp_df

    #Append/add new motors
    else:
        X_test_df = X_test_df.append(X_test_temp_df)
        X_train_df = X_train_df.append(X_train_temp_df)

    #Convert X and Y to type array
    X_train = X_train_df.values
    Y_train = np.array(Y_train)

    X_test = X_test_df.values
    Y_test = np.array(Y_test)

    #Transfor Y train ans test dimensions from [number of examples, ] to
[number og examples, 1]
    Y_train = Y_train.reshape((Y_train.shape[0],1))
    Y_test = Y_test.reshape((Y_test.shape[0],1))

    return X_train, Y_train, X_test, Y_test
```

```
#Standardscaler
def standardize(data):
    d = preprocessing.StandardScaler().fit_transform(data)
    return d
```

```
# Load treated data
def load_treated_data(num_motor):

    #Load train- and testset
    train_x, train_y, test_x, test_y = load_n_motors(num_motor)

    #Normalize the values in X train and test
    train_norm_x = normalize(train_x)
    test_norm_x = normalize(test_x)

    #Run the normalized values through a robust scaler
    train_norm_rob_x = robust_scaler(train_norm_x)
    test_norm_rob_x = robust_scaler(test_norm_x)

    #Delete columns with std lower than ...
    #train_norm_rob_x, test_norm_rob_x = del_col(train_norm_rob_x, test_
norm_rob_x)

    return train_norm_rob_x.T, train_y, test_norm_rob_x.T , test_y
```

Code from "NN_2_Layer.ipynb"

```
#Import numpy for array treatment and matrix multiplication
import numpy as np

#Import sklearn for mathematical computation of MSE
import sklearn
from sklearn.metrics import mean_squared_error

#Import matplotlib to visualize results
import matplotlib.pyplot as plt

#Import own code for data-treatment and to load training and test set
from Data_treatment import *
```

```
#Initialize Parameters W1, b1, W2 and b2 as numpy arrays
#Input: n_x: #features from training set | n_h: #hidden layers | n_y: number of nodes in output layer
def initialize_parameters(n_x, n_h, n_y):

    #Initialize numpy arrays
    W1 = np.random.randn(n_h, n_x)*0.1 # Numpy array of dimension [n_h, n_x] with random numbers
    b1 = np.zeros((n_h, 1)) # Numpy array of dimension [n_h, 1] with zeros
    W2 = np.random.randn(n_y, n_h)*0.1 # Numpy array of dimension [n_y, n_h] with random numbers
    b2 = np.zeros((n_y, 1)) # Numpy array of dimension [n_y, 1] with zeros
    return W1, b1, W2, b2
```

```
#Forward propagation over one layer: Calculating Z and A(Z)
def forward_prop(W,b,A_prev,a_function):
    Z = np.dot(W,A_prev)+b #For first layer, ex: Z = W * X + b

    if a_function == "Sigmoid":
        A = 1/(1+np.exp(-Z)) #A = Sigmoid(Z)

    elif a_function == "None":
        A = Z #No activation function in the last layer gives A = Z
    return A, Z
```

```
# Takes in Y and the predicted value from NN and forward propagation,
# and calculates the cost with Mean Squared Error(MSE)cost function
#Both Y and Yhat are numpy arrays of dimensions [number of examples, 1]
def cost_func(Y, Yhat, m):

    #MSE cost function
    cost = (1/(2*m)) * np.sum((Yhat-Y.T)**2)

    cost = np.squeeze(cost) # Turns the dimension of cost from [[17]] into 17).
    assert(cost.shape == ())
    return cost
```

```

def backprob(X, Y, A1, Yhat, W2, m):

    #Part derivatives
    dCost_dYhat = (Yhat-Y.T) #dCost/dYhat
    dYhat_dW2 = A1 #dYhat/dW2
    dYhat_dA1 = W2 #dYhat/dA1
    dA1_dZ1 = A1*(1-A1) #dA1/dZ1 = Derivative of sigmoid(Z1)
    dZ1_dW1 = X #dZ1/dW1

    #Calculating dW1 and db1
    dW2 = (1/m) * np.dot(dCost_dYhat, dYhat_dW2.T)
    db2 = (1/m) * np.sum(dCost_dYhat, axis=1, keepdims=True)

    #Calculating dW1 and db1
    dW1 = (1/m) * np.dot((np.dot(dYhat_dA1.T,dCost_dYhat) * dA1_dZ1),X.T
)
    db1 = (1/m) * np.sum((np.dot(dYhat_dA1.T,dCost_dYhat) * dA1_dZ1), ax
is=1, keepdims=True)

    return dW1, db1, dW2, db2
    
```

```

def model(X, Y, learning_rate, epoch):

    # Set the number of neurons in each layer
    n_x = X.shape[0] #Input layer
    n_h = 2 #Hidden layer
    n_y = 1 #Output layer

    m = Y.shape[0] # number of examples
    cp = np.zeros((epoch-1)) #Numpy array for storing cost from each epo
ch

    #Initialize parameters
    W1, b1, W2, b2 = initialize_parameters(n_x, n_h, n_y)

    #Iterate and update parameters epoch number of times
    for i in range(1,epoch):

        #Forward propagation through two layers,
        A1, Z1 = forward_prop(W1, b1, X, "Sigmoid") #Sigmoid activation
        in the first/hidden layer
        A2, Z2 = forward_prop(W2, b2, A1, "None") #No activation in t
he second/output layer

        #Cost function: Mean sqaured error
        cost = cost_func(Y,A2,m)

        #Register cost for plot
        cp[i-1] = cost

        #Backward propagation, finding the weigth and biases
        dW1, db1, dW2, db2 = backprob(X, Y, A1, A2, W2, m)
    
```



```
#Update Parameters
W1 = W1 - learning_rate *dW1
W2 = W2 - learning_rate *dW2

b1 = b1 - learning_rate *db1
b2 = b2 - learning_rate *db2

parameters = [W1, b1, W2, b2]

# summarize history for loss
plt.plot(cp)
#plt.plot(history.history['accuracy'])
#plt.plot(scores)
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train'], loc='upper left')
plt.show()

print("Final Cost: ", cost)
print("Final Squared Cost: ", np.sqrt(cost))

return parameters
```

```
#Set hyper parameters
learning_rate = 0.012
epoch = 500
num_motor=100

X, Y, Xt, Yt = load_treated_data(num_motor)
#print_shapes(X,Y,Xt,Yt)
```

```
# Run Model 1
parameters = model(X, Y, learning_rate, epoch)
```

```
#Define function to test trained NN on Xt and Yt
def test(parameters, Xt, Yt):
    # Set the number of neurons in each layer
    n_x = Xt.shape[0]    #Input layer number of features or input neurons

    n_h = 2              #Hidden layer
    n_y = 1              #Output layer

    m = Yt.shape[0]    # number of examples

    #Initialize parameters
    W1, b1, W2, b2 = parameters

    #Forward propagation through two layers,
    A1, Z1 = forward_prop(W1, b1, Xt, "Sigmoid")    #Sigmoid activation in
    the first/hidden layer
    A2, Z2 = forward_prop(W2, b2, A1, "None")    #No activation in the s
    econd/output layer

    #Cost function: Mean sqaured error
    c = (Yt.T-A2)
    d = np.sqrt((Yt.T-A2)**2)
    cost = (1/m)*np.sum(np.sqrt((A2-Yt.T)**2))

    print("Final Cost: ", cost)
    return cost
```

```
# Run test on Model 1
cost = test(parameters, Xt, Yt)
```

Code from “Keras.ipynb”

```

#Import Keras to build and train ML model
import keras
from keras import optimizers
from keras.models import Sequential
from keras.layers import Dense

#Import sklearn fir mathematical computation of MSE of pred-Y
import sklearn
from sklearn import metrics

#Import matplotlib for visualisation of results
import matplotlib
from matplotlib import pyplot as plt

#Import own kode for datatreatment and to load training and test set
from Data_treatment import *

#Ser hyperparemeters
num_motors = 100
batch_size = 128
learning_rate = 0.01
epochs = 25

#Load datasets for training and testing
X, Y, Xt, Yt = load_treated_data(num_motors)

#print_shapes(X,Y,Xt,Yt)
#print("Number of features: " + str(X.shape[1]))

X = X.T
Xt = Xt.T
#print_shapes(X,Y,Xt,Yt)
#print("Number of features: " + str(X.shape[1]))
    
```

```

# create model input format: [num_ex, num_features]
model = Sequential()

#First hidden layer
model.add(Dense(2,
                input_dim=X.shape[1], # Features from input X
                kernel_initializer='random_uniform', #Initialize W
                bias_initializer='zeros', #Initialize b
                activation='sigmoid' #Activation function
                ))

#Output layer
model.add(Dense(1,
                kernel_initializer='random_uniform',
                bias_initializer='zeros'
                ))
    
```

```
#Define optimizer: Only learning rate is set.
sgd = optimizers.SGD(lr=learning_rate)

#Compile model with loss function and optimizer
model.compile(loss='mean_squared_error', optimizer = 'sgd')

# Fit/ train the model
history = model.fit(X, Y, epochs= epochs, batch_size= batch_size, verbose=0)

print('Train lowest mse: ' + str(min(history.history['loss'])))
print('Train lowest sqrt(mse): ' + str(np.sqrt(min(history.history['loss']))))
```

```
# Summarize history for loss
plt.plot(history.history['loss'])
#plt.plot(history.history['accuracy'])
#plt.plot(scores)
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

#print(history.history['mean_squared_error'])
print('Train lowest mse: ' + str(min(history.history['loss'])))
print('Train lowest sqrt(mse): ' + str(np.sqrt(min(history.history['loss']))))
```

```
#predict:
pred = model.predict(Xt)

# evaluate the model
score = metrics.mean_squared_error(pred, Yt)
print("Score: ", score)

score_root = np.sqrt(score)
print("Rooted score: ", score_root)
```