Design of a Neural Network for State-of-Charge Estimation

Hochschule Bochum
Bochum University
of Applied Sciences

by

Fatih Koc
017200607

Submitted for the degree of

Bachelor of Science

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

FIRST SUPERVISOR: PROF. DR.-ING. EDMUND COERSMEIER
SECOND SUPERVISOR: DIPL.-ING. (FH) ANDREAS KOCH

May 2022

Name:	Fatih Koc
Matriculation ID:	017200607
E-mail:	fatih.koc-@hotmail.com
	fatih.koc-@outlook.com
Study program:	B.Sc. Computer Science
University:	Hochschule Bochum / Bochum University of Applied Sciences
Processing time:	2 months
First supervisor:	Prof. Dr.-Ing. Edmund Coersmeier
Second supervisor:	Dipl.-Ing. (FH) Andreas Koch

Abstract

The usage of batteries in recent years become widespread in many fields. In addition to the classic tasks such as mobile and IoT devices, they enable off-grid systems, energy renewable systems, standalone systems and more. Especially, since the worldwide awareness of environmental issues and oil shortage, electric vehicles has become inevitable. Such systems often require a robust approach for estimation of the State-of-Charge (SOC) not only to protect the battery from damages, but to guarantee safe operations. Over the decades, a lot of methods have been discovered, developed and used to predict the SOC. It turned out that adaptive systems do the best, in point of accuracy and efficiency. Adaptive systems are characterized by the ability of designing themselves, and therefore they can automatically adjust in changing systems, which is why they are so powerful. This work presents the full design (theoretical and practical) of an artificial neural network. Instead of using the classic SOC as target value, a new normalized SOC is defined as a simplification approach for practical purposes. The design steps used in this work are generally valid for all feed-forward artificial neural networks. Nevertheless, many steps are also be transferable to other models and domains as well. The Bayesian optimization strategy is used to globally find a good hyperparameter set.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der in den beigefügten Verzeichnissen angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Der Durchführung einer elektronischen Plagiatsprüfung stimme ich hiermit zu. Die eingereichte elektronische Fassung der Arbeit entspricht der eingereichten schriftlichen Fassung exakt. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen hat.

Bochum, 22.05.2022
Ort, Datum

[illegible]
Unterschrift

TABLE OF CONTENTS

LIST OF TABLES

FIGURE TABLE

GLOSSARY

ADAM Adam 'adaptive moment' optimizer.

ANN Artificial Neural Network.

BO Bayesian Optimization.

DNN Deep Neural Network.

DOD Depth-of-Discharge.

EI Expected Improvement.

EIS Electrochemical Impedance Spectroscopy.

ELU Exponential Linear Unit.

GP Gaussian Process.

KF Kalman Filter.

ML Machine Learning.

NN Neural Network.

OCV Open-Circuit Voltage.

ReLU Rectified Linear Units.

SGD Stochastic Gradient Descent.

SOC State-of-Charge.

SOC$_H$ State-of-Charge with consideration of SOH.

SOC$_N$ normalized State-of-Charge.

SOF State-of-Function.

SOH State-of-Health.

SVR Support Vector Regression.

tanh Hyperbolic Tangent Function.

Chapter 1

Introduction

1.1 Motivation

The usage of batteries in recent years become widespread in many fields. In addition to the classic tasks such as mobile and IoT devices, they enable off-grid systems, energy renewable systems, standalone systems and more. Especially, since the worldwide awareness of environmental issues and oil shortage, electric vehicles has become inevitable. As important as these systems are, an unreliable battery can lead to unpredictable defects or even total failure of systems. It is often needed to understanding and to monitoring the cells states at a particular point in time to protect the battery from damages and to ensure functionality and safety of the system. The State-of-Charge (SOC) (see 1.2.2) of the battery is one of the most important parameters, and an accurate knowledge of this quantity is necessary for all battery management tasks. Roughly speaking, it indicates how much energy the battery is left with.

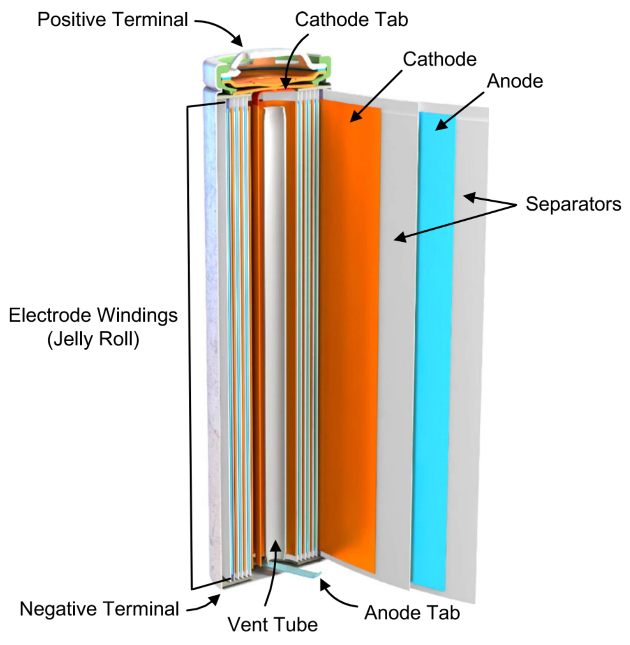

Figure 1.1: A typical 18650 Li-ion cell

Source: [Nag+21] Licensed by: link.springer.com

A battery is a chemical energy concentration covered by a case. figure 1.1 shows the structure of a typical 18650 Li-ion cell. The core parts of the battery are the electrodes, which are called cathode and anode. Both electrodes have different chemical potentials from each other and the property to store energy. During discharge, electrons flow from the anode to the cathode through an external circuit, providing power to that circuit. However, since the charge carriers are not accessible, it is not possible to measure the SoC directly by existing sensors. Other quantities such as battery voltage and current can be easily measured. Therefore, several difficult methods were developed to bypass this issue. Some works and literature categorizes the methods in their work. The section 1.3 gives a brief overview of these methods. Before that, some important definitions for this work are discussed in section 1.2. These are used to summarize and discuss the objectives of this work in 1.4.

1.2 Definitions Of Different Battery Properties

Important definitions for cell state specifications are discussed in this section. 1.2.1 introducing the SOH. This is the basis for the SOC, defined in 1.2.2. A problem of these definitions is discussed in section 1.2.3 and a more general approach for cell state indication is shown.

1.2.1 Definition of State-of-Health

The state of health (SOH) indicates the condition of a battery, compared to its ideal condition. The units of SOH are percent points. [1][MG] the SOH is defined as

$$SOH(t) = \frac{Q_{max}(t)}{Q_n}(*100\%) \tag{1.1}$$

with $Q_{max}(t)$ the maximal releasable capacity at time t and Q_n the nominal capacity or manufacturing capacity. The longer a battery is used, the lower the SOH.

1.2.2 Definition of State-of-Charge and Depth-of-Discharge

The State-of-Charge (SOC) of a battery is defined as the ratio of its current capacity $Q(t)$ to the nominal capacity Q_n and is an expression of the present battery capacity, as defined in equation 1.2. Even if the equation 1.2 seems to be clear, the meaning of Q_n various in different literature. Some (probably the larger proportion) see Q_n as a constant or specification published by the manufacturer for a specific battery type, set during the design process. This definition was also used in 1.1. Others see it as a variable, which is influenced by the SOH. [HG17] This work differentiate both views as defined in equation 1.2 and equation 1.3, made identifiable by the label H and with Q_n as manufacturer designed capacity of the battery. In some scenarios, the expression of Depth-of-Discharge (DOD) is used alternatively as the reverse to SOC, see equation 1.4. As the name suggests, the DOD describes how deeply a battery has been discharged compared to its maximum capacity. The units of all are percent points.

$$SOC(t) = \frac{Q(t)}{Qn}(*100\%) \tag{1.2}$$

$$SOC_H(t) = \frac{Q(t)}{SOH * Qn}(*100\%) \tag{1.3}$$

$$DOD(t) = 1 - SOC(t) \tag{1.4}$$

1.2.3 Defining normalized State-of-Charge

The value of the SOC (and SOC_H) varies between 0% and 100%. If the SOC is 100%, then the cell is said to be fully charged, whereas a SOC of 0% indicates that the cell is completely discharged. However, in practical applications, as a cell starts ageing, the maximum SOC starts to decrease. This means that for an aged cell, a 100% SOC would be equivalent to an e.g. 80% SOC of a new cell. [GM17, Chapter 7, SOC] In other words, the SOH influences the SOC. The SOC_H can overcome this representation problem, since it includes the effect of the SOH. Similarly, the SOC is not allowed to fall below a certain threshold (e.g. 30%), because it would cause battery damages and/or some percentage of the energy needs to be reserved for other operations e.g. hybrid-work, security, engine start etc. Therefore, the cell has to be recharged when the SOC reaches the threshold. In all these cases, the actual range of the SOC changes drastically, which even the SOC_H

can not represent. Additional complications occur, due to environmental influences on the battery, like temperature. [Gro] For this purpose, the normalized State-of-Charge (SOC_N) is introduced, made identifiable by the label *N*. To the best of my knowledge, this or something similar has not yet been introduced.

The basis for this approach is the min-max scaler rule:

$$\mathbf{x}' = \lambda_1 + \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})}(\lambda_2 - \lambda_1) \tag{1.5}$$

where $\mathbf{x}'$ is the rescaled form of $\mathbf{x}$ with the scaling bounds λ_1 and λ_2. To further generalize the SOC_H, the following substitution of 1.6 for equation 1.5 and 1.7 for equation 1.3 are applied:

$$\mathbf{x} = Q \tag{1.6}$$

$$SOH * Q_n = max(Q) \tag{1.7}$$

The λ_1 and λ_2 fall away for a defined SOC range of 0%-100%. If the substituted equation 1.3 is transferred to the rule 1.5, the normalized State-of-Charge (SOC_N) is then defined as:

$$SOC_N(t) = \frac{Q_k(t) - min(Q_k)}{max(Q_k) - min(Q_k)}(*100\%) \tag{1.8}$$

where $min(Q_k)$ and $max(Q_k)$ form the boundaries for an application. The data is typically divided into subsets in data-driven (discussed below 1.3.3) approaches, e.g. for subdivision into temperatures. Index *k=1,2,3,...n* denotes each subset of the data. The SOC_N is applied in 2.3.1.

1.3 Brief Overview Of SoC Estimation Methods

The SOC estimation methods are categorized in some literatures. [WBM12][Zho+21] The recent and most complete work of Wenlu et al. [Zho+21] classifies the SOC estimation process first in *Battery modelling*, with respect to the SOC estimation as:

- electrochemical mechanism models (1.3.1)
- equivalent circuit models (1.3.2)
- data-driven models (1.3.3)

and secondly, separating them through different *algorithm types* as:

- direct measurement (1.3.4)
- black-box battery model (1.3.5)
- state-space models (1.3.6)

To increase the accuracy or to get other benefits, combinations of these algorithms are possible.

1.3.1 Electrochemical Mechanism Models

Electrochemical mechanism models, such as pseudo-two-dimensional models (P2D), single-particle models (SP) and the simplifications of them aim to describe and simulate the inner workings of the battery through equations and large calculation times. Thereby, one is able to map the internal and external characteristics of a battery at the same time to a specific battery. This technique can result in very high accuracy, but got the risk of poor adaptability to some working conditions. Moreover, choosing and implementing the model requires the identification of parameters and prior knowledge about the batteries.

1.3.2 Equivalent Circuit Models

Similar to electrochemical mechanism models, the equivalent circuit models using techniques to simulation the static and dynamic characteristics of a battery. The difference here is the modelling through circuit networks and a basic constant voltage source to get nearly the same behaviour as the characteristics of a battery. One challenge here is to keep the amount of circuit components so low as possible, and at the same time to get a sufficient accuracy. The equivalent circuit models require the identification of parameters as well.

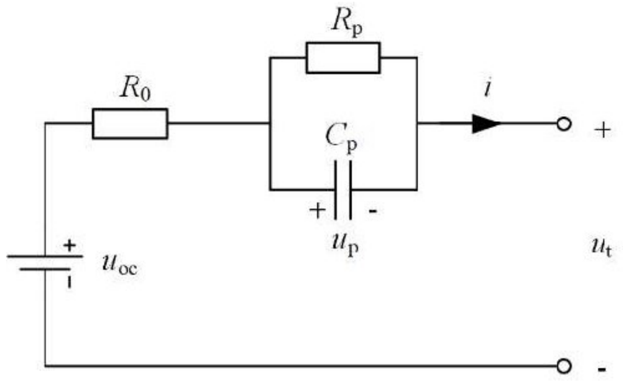

Figure 1.2: Example equivalent circuit: Thevenin model - using resistor R_0 as battery internal resistor and an R||C network to simulate electrochemical polarization effects of the battery. U_{OC} is an ideal voltage source.

Source: [XZa19]

1.3.3 Data-Driven Models

The core of data-driven models is the collection of data about a specific battery, that usually have to be collected during experiments. The main concept of data-driven models are to find relationships between the input and output data, where the data is non-linear in most cases and without the specific knowledge of the physical behaviour of the system. Data-driven modelling do not need as much knowledge about the specific battery compared to equivalent circuit models and in particular electrochemical mechanism models, as the collected data can approximate the behaviour of the battery. Data-driven models rely on integrity of the data. If they can not depict the battery correct, the accuracy will be negatively affected in practice. The estimation accuracy depends heavily on the number of samples, which is why generalization problems can occur, see 3.1.4. That are the reasons why data collection is a difficult task. Main advantages are the omission of modelling and parameterization, and moreover the data-driven methods are totally battery type independent. These types of models are also called (complex) adaptive systems. They are adaptive because they show a special ability to adapt to their environment and have the possibility to learn (from experience).

1.3.4 Direct measurement

Direct measurement methods are typically inaccurate but cheap methods of SOC estimation, and they do not need much computing power. They use some battery physical attributes, such as Open circuit voltage (OCV), terminal voltage, resistances and impedance measurements, or by counting the flowed current. These battery parameters are mostly relatively easy to measure. The online measurement resulting in an approximate value of the SOC. Especially popular is the method of flowing current, which is usually called *coulomb counting* or *ampere-hour integration* and is a very old SoC estimation method. In the simplest version, the SOC is calculated by the following equation:

$$SOC(t) = SOC(t_0) \pm \int_{t_0}^{t} \frac{I(\tau) \cdot \eta(\tau)}{Qn} \, d\tau \qquad \tau \in [t_0, t] \tag{1.9}$$

where $SOC(t)$ is the SOC of the battery at time t, $SOC(t_0)$ is the initial SOC of the battery, and $I(\tau)$ and η are the battery current and efficiency factor over the time t_0 to t. Coulomb counting is efficient in terms of computing power, but the accuracy is affected by factors such as temperature, battery history and time dependent characteristics. More drastically, the cumulative effect introduced by integration and approximation makes the estimation sensitive to measurement errors due to noise, resolution and rounding. This error can get larger over time. The modified version of the coulomb counting method is able to increase the accuracy further, and is able to compensate the above factors. Application of modified coulomb counting in SOC estimation can be found in [Bac+17]. The principle of modified coulomb counting is the same as the usual coulomb counting, but it uses a corrected current instead of the basic current for the calculations. Some correction factors must be determined with experiments and with the regular Coulomb counting method.

The OCV method is an alternative which depends on fewer factors. The OCV of a battery is the voltage between the battery terminals without any load. There is approximately a linear relationship between the SOC and OCV, which is used to estimate the SOC. But this method is not working for all battery types, since certain type of batteries does not have a linear relationship between the OCV and SOC. Additionally, the battery needs to rest for a long time to archive the linearity. This method don't require many computational resources, since it can be implemented with look-up tables or curve fitting functions.

Electrochemical Impedance Spectroscopy (EIS) is an electrochemical technique to study the impedance of a battery. Thereby, a large range of AC frequencies are sent to a battery. At different frequencies, different phenomenon happening inside the battery. From these phenomena, one can gain different parameters, which can be correlated to the SOC. It is possible, to get high accuracies with this method, but one problem of this estimation method is that the estimation accuracy is strongly influenced by the temperature. A second problem comes with the impedance increase with the age of the batteries.

1.3.5 Black-box Battery model

Black-box battery models use the principle of data-driven models 1.3.3. The Battery is treated as an unknown system in this method. Data and Information is collected from the battery, and divided in input and output. Inputs can be voltage, current, temperature, etc. and output can be SOC, SOH etc. A model is then trained, which is then supposed to map and interpolate all possible values. Illustration can be found in figure 1.3.

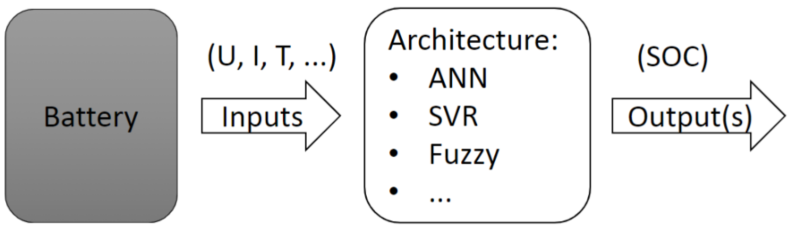

Figure 1.3: Black-box principle with inputs: voltage(U), current(I), temperature(T) and output: SOC

Source: own

Different model architecture approaches can be found in the literature for input-output relationship mapping, including artificial neural networks (ANN), support vector regression (SVR) and fuzzy algorithms. ANNs are built by artificial neurons, with a paradigm directly inspired by and interconnected like a brain. They contain one or more neurons, and are ordered in layers. A network is created by connecting these layers. There are many types/architectures of ANNs. Some examples are neural networks (NN), recurrent neural networks (RNN) and convolutional neural networks (CNN). The simplest is the NN, consisting of exactly three layers, the input layer, the hidden layer, and the output layer, see figure figure 1.4.

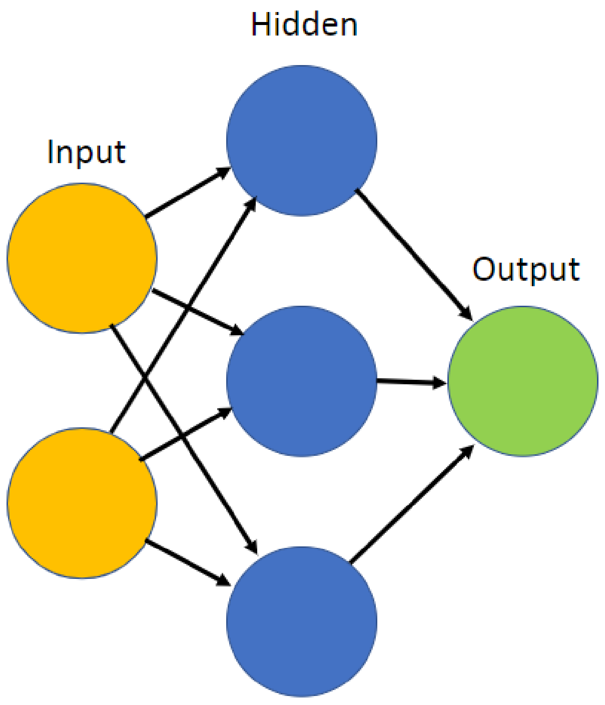

Figure 1.4: Neural Network with input layer (yellow), hidden layer (blue) and output layer (green) in a (2-3-1) neuron constellation.

Source: own

ANNs can be either shallow or deep. They are called shallow when they have only one hidden layer, or deep when hidden layers are more than one. Deep networks often perform generally better at the cost of longer training time and more complex architecture. In general, as long as training data amount are enough and achieving a satisfactory prediction accuracy, it can be used for SOC prediction. ANNs are explained in more detail in the chapter 3.1.

The objective of SVR algorithms is to solve regression fitting-problems in an n-dimensional space by finding the best hyperplane equation in terms of covering data points. Many regression problems like the SOC cannot be regressed linearly, which is why it is necessary to use a transformation method in which a kernel is used to increase the dimension of the data-space. [Bla+13, section SVM Method] ANNs as well as SVR have good generalization abilities 3.1.4.

The hurdle of this type of models is to deal with hyperparameter (see 3.2) tuning like the number of layer or neurons ANN types or specific for SVR, the choice of the best fitting kernel for example. The SOC estimation accuracy and the generalization ability of a model is heavily dependent on the choice of these parameters. Hyperparameter tuning is a particular problem, since there does not exist an optimal algorithm or mathematical way to determine the best set. The current only way to solve these problems is through trial and error processes.

1.3.6 State-space Models

The state space representation of a system is a mathematical model, which describes the system with a (first-order differential) equations for its output in relation to the input and state variables. A state in the context of state space analysis is the summarization of the history by a group of variables. This state is used for prediction tasks such as the SOC. In case of a SOC estimation, the SOC is used as feedback for the model and is linked with the input variables. The two main components of the model are the battery model and the estimation algorithm. For scheme, see figure 1.5

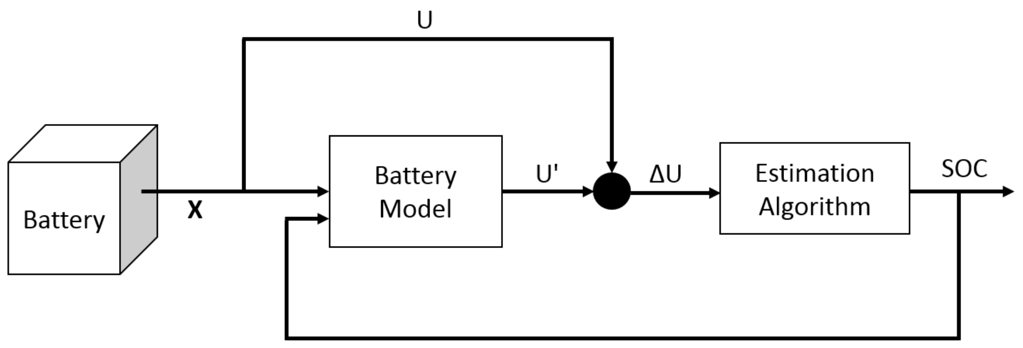

Figure 1.5: State-space model-based estimation scheme for SOC estimation with input vector x and the actual SOC as output. Battery voltage U and Battery Model voltage U' are used to estimate the SOC as example. In general, all parameters are possible that have a direct relationship with the SOC [CF10]

Source: own, inspired by [Tin+14, Fig.1] [Zho+21, Fig.6]

Different Battery Model categories are already presented in chapter electrochemical mechanism models (1.3.1), equivalent circuit models (1.3.2) and data-driven models (1.3.3). The accuracy of the battery model can greatly affect the SOC estimation accuracy, as the estimation algorithm depends on it. Several estimation algorithms are proposed in literature: H∞ filter, sliding mode observer, statistical filter algorithm, PI observer, improved particle filter algorithm, proportional integral observer, Luenberger observer and the one of the most frequently adopted method - the Kalman filter (and its extensions).

The Kalman filter (KF) is an algorithm that uses multiple input sources instead of a single input alone to archive a more accurate estimation of the unknown variable, like the SOC. The goal is to filter possible statistical noise, uncertainties and other inaccuracies, by combining the probability distributions of all sources. The general KF works in mainly two phases, the prediction and correction. At first, the KF produces an estimation of the system state along with its covariance matrix. Then the *Kalman Gain* can be calculated, by *a priori* estimate covariance and the measured quantities' covariance, which indicates the trust-weight of the measurement versus prediction. This calculation is used to update the *a posteriori* state estimation and the *a posteriori* covariance. One of the most important properties of the KF is the linearity of the system, i.e. the prediction and correction step both will contain Linear functions only. Since the battery SOC is not linear, the basic KF usually gives poor results. Therefore, other extensions for the KF are developed. Probably, the extended Kalman filter is the most popular one. The main difference from the extended KF is the linearized approximation of the variables with the usage of Taylor series expansion. The extended KF method is suitable for all types of batteries, and it could accurately estimate the battery SOC whose current is rapidly changing. [Tin+14] [Zho+21] [Mon11]

1.4 About This Work

As indicated in 1.1, an accurate estimation of the SOC is essential for BMS systems. Therefore, to solve this issue, different paradigms are proposed and used to estimate the battery state as presented in 1.3. One of the most popular ways for SOC estimation are the Black-box battery models 1.3.5, for various reasons, as described in the respective section. However, a problem is indicated in that section, where the determination of hyperparameters can only be done by trial and error processes. Therefore, the performance of the models can only be roughly compared because the implementation circumstances are different and the way the results are achieved is often not obvious. This work is intended to provide a basis for data-driven implementation tasks specific for SOC estimation, but can also be used for other data-driven fields. An automatic optimization strategy is elaborated to optimize hyperparameters human independent. Further, the introduced normalized State-of-Charge (SOC_N) is used in this work, defined in section 1.2.3.

The research objectives concretely are

- An accurate battery capacity estimation, using the new defined SOC_N
- Design process of a Black-box battery model, especially for SOC estimation
- An automatic hyperparameter optimization strategy is elaborated

Only a (deep) neural network take the algorithmic role in the practical part of this work, as the scope of the work has to be narrowed down. Nevertheless, many steps are also be transferable to other models as well. Accordingly, the thesis contents are briefly introduced in figure 1.6.

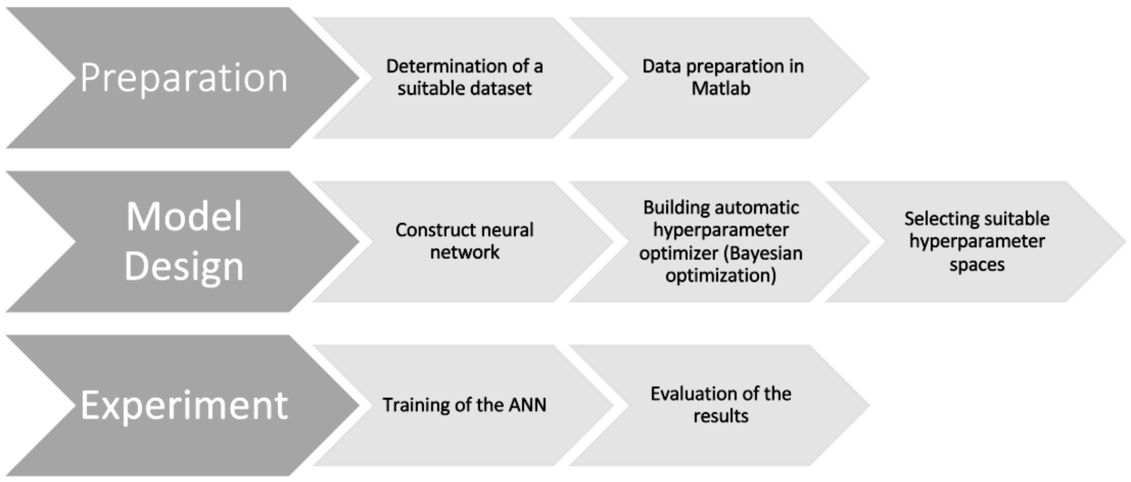
Preparation
Determination of a suitable dataset
Data preparation in Matlab
Model Design
Construct neural network
Building automatic hyperparameter optimizer (Bayesian optimization)
Selecting suitable hyperparameter spaces
Experiment
Training of the ANN
Evaluation of the results

Figure 1.6: Working procedure for SOC estimation approach

Chapter 2

Preparation Tasks

The aim of this chapter is to describe preparation works, before a model can be designed and trained. This includes in first step to set up the environment in section 2.1 for replication purposes. The used system specifications for the experiments are displayed in subsection 2.1.1, followed by the description of how the tools and environment is setted up for the practical part of this work in 2.1.2. This finds usage for the task described in the following section, where the focus is on data preparation for the machine learning process. For this purpose, a suitable dataset is first selected in section 2.2 and thereafter the selected dataset's preprocessing steps are discussed.

2.1 Technical Environment

2.1.1 System specifications

The hardware and software used in the experiments is listed below.

1. Hardware

CPU :	Intel Core i7-4600U CPU @ 2.10GHz
GPU :	-
RAM :	7,6 GiB

2. Software

OS :	Manjaro Linux 21.2rc2 Qonos
Kernel :	Linux 5.13.19-2 Kernel
Matlab :	R2021a
Python :	3.9.7
Keras :	2.4.3
Tensorflow :	2.4.1
Tensorboard :	2.4.0

Gpy : 1.10.0

Gpyopt : 1.2.6

Conda : 4.10.3

2.1.2 Setting up the Environment

For preparation tasks of the dataset, the software *Matlab* is used, which has the ability to automatically evaluate and calculate numerical data in large amounts at high speed. Furthermore, the combination of user interfaces and command code increases usability. The platform-independent tool can be downloaded from here and the installation is straightforward:

`https://de.mathworks.com/downloads`

The environment manager *Conda* is used to simplify package installations and to avoid possible dependency problems. The following script is used for the installation:

```
$ wget https://repo.anaconda.com/archive/
Anaconda3-2021.05-Linux-x86_64.sh
```

Hash-verification of SHA-256 checksum and run the script

```
$ sha256sum Anaconda3-2021.05-Linux-x86_64.sh
$ bash Anaconda3-2019.03-Linux-x86_64.sh
```

Installation success and all installed packages can be seen with

```
$ conda --version
$ conda list
```

Creating a new conda environment named *soc_estimator* and activate it via

```
$ conda create --name soc_estimator
$ conda activate soc_estimator
```

The popular high-level API *Keras* in combination with the backend framework *Tensorflow* is used to build and train the machine learning model. Keras is well suited for prototyping as it offers the possibility to experiment with network architectures easily and quickly. Keras creates another abstraction layer between the machine learning backends, such as Tensorflow, and the user, which makes it possible to create a powerful ANN with only a few lines of code. Since Keras compiles the ANN internally to a Tensorflow-graph, Keras still brings the flexibility of Tensorflow. Tensorflow is a machine learning framework from Google written in Python, C++ and CUDA and brings APIs for Python and others. It is suitable for a wide range of platforms for instance Linux and Windows, and for mobile/IoT devices is tensorflow lite available. Since Python is a (hybrid) interpreted language, it makes sense to perform the computationally intensive operations in a more efficient language. The Tensorflow packages providing a built-in interface for this task. [3][4] Tensorflow and Keras can be installed with following lines

```
$ conda install tensorflow
$ conda install keras
```

Note that the following section has been brought forward for completeness and see 3.2 for the respective chapter. For the architecture optimization tasks, the library GPy/GpyOpt is used, which is a Bayesian optimizer package. Alternative software can be found here [Fra18, chapter 6]. GPy is a big and powerful Python package with many features for Gaussian process modelling. GpyOpt is based on GPy, specialized for Bayesian optimization. GpyOpt is less popular then the alternatives scikit-optimize or Optuna for unknown reason. However, the main advantage of GpyOpt is the modularity that it provides, which makes it flexible to use. The packages from GPy allowing further changes and modifications of the modules. It should be noted that the maintanace of GpyOpt ended. GPy and GPyOpt can be installed with

```
$ conda install -c conda-forge gpy
$ conda install -c conda-forge gpyopt
```

Other useful conda commands are bellow. First shows all conda environments and second allows the deactivation of the active environment.

```
$ conda info --envs
$ conda deactivate
```

2.2 The Used Dataset

Maschine Learning based approaches require large amounts of data for training and perform poorly when predicting ‘out of distribution’. [GBC16, pp. 1.2.2, 1.2.3] [Dem20, page 26] High quality data is an important prerequisite for the quality of predictive models. To avoid poor performance results of the model due to bad initial data, it is essential to investigate the data in order to detect problems at an early stage and to be able to take appropriate steps for preprocessing and cleaning.

One can check the overall quality of the data by checking the following:

- The number of data point records.
- The number of features/attributes.
- The number of missing values in the dataset.
- Inconsistent data points e.g. check the permissible value ranges.

Since the topic of SOC prediction has existed for a long time, there are also numerous data sets to choose from. The work of *Reis et al.* [Rei+21] is used to obtain an overview and to select an appropriate set. An example of an unchoosen dataset is the *NASA Battery Data Set*, see A for more information. The spezific requirements for the SOC estimation dataset, in addition to the quality criteria stated above, are as follows:

- The data set must contain the most basic features including voltage, current, temperature and ideally electric charge
- The tests were carried out in a wide temperature range and maximum possible drive cycles (explained below)

According to the criteria above, the selected dataset for this work is for the 2.9 Ah NCA Panasonic 18650PF Li-ion cell provided by University of Wisconsin-Madison, the dataset can be downloaded from [5]. The cell was cycled systematically through five different ambient temperature ranges, containing +25°C, +10°C, 0°C, -10°C and -20°C. The primary target of the dataset is the automotive industry, therefore this range of temperature values is suitable for working under real conditions. Standard test procedures, so-called *drive cycles*, are used to ensure that the data reflects the correct behavior of the battery. A driving cycle is a standardized series of data points representing the speed of a vehicle versus time. This drive schedules are used to generate cell usage profiles. Wisconsin-Madison has included 9 driving cycle profiles for each of the five ambient temperatures. This means that a total of 45 data profiles are available for the experiments. The dataset includes in-cycle measurements of terminal voltage, current, cell temperature, test chamber (ambient) temperature, capacity, energy, measured watt. Additionally, the time was recorded, starting with zero at the beginning of every cycle until the end of the cycle. Further EIS impedance readings and Hybrid Power Pulse Characterisation (HPPC) are included, which are not in the interest of this work. The dataset with all profiles has 3,675,599 data points and is provided in '.mat' format. [5, see readme file] A plot of example data points from the dataset can be seen in figure 2.1.

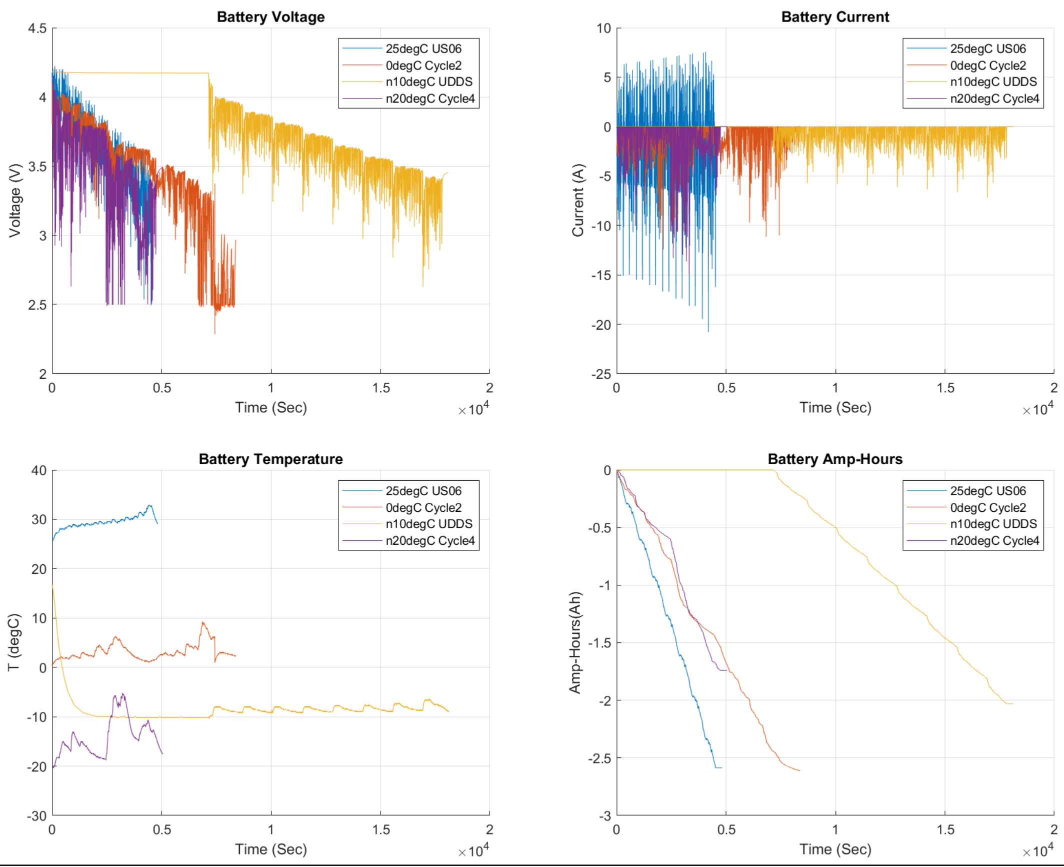

Figure 2.1: The original 2.9 Ah NCA Panasonic 18650PF Li-ion cell dataset provided by University of Wisconsin-Madison is plotted here, the download link [5]. The features and target are plotted over time in seconds. Features are terminal voltage, current and cell temperature and the target is amp-hours. Four examples are depicted from different drive cycles and temperatures. Blue is from cycle *US06* and 25°C. Red is from cycle *Cycle2* and 0°C. Yellow is from cycle *UDOS* and -10°C. Purple is from cycle *Cycle4* and -20°C

Source: own

2.3 Data Preparation

Preprocessing and cleaning of data are important tasks that must be performed before a dataset can be used to train models. Unformatted data often contains unnecessary or missing values and is unreliable. Furthermore, real data may contain irregularities or corrupted data that affect the quality of the dataset. Using this data for modeling can lead to incorrect results and poor performance. The following quality issues are more common with data:

Incomplete information: The data is missing attributes or values.
Superfluous information: The data contains erroneous records or outliers.
Inconsistent information: The data contains conflicting records or discrepancies. [Dem20, p.33]

When problems are identified for the data, preparation steps are required. These often include cleaning up values, data transformation and more. However, no specific procedure is clearly established in the literature. Therefore, not all the data preparation steps are always necessary for an application and some are experimental. For this work only the essential necessary steps are implemented and each of the used ones are explained in its respective sections. All preparation steps were done with the scientific tool *Matlab*. An overview of the applied preprocessing steps can be seen in figure 2.2 and the respective code for every implemented step can be found in appendix B. The steps and some further additions are discussed in the following subsections.

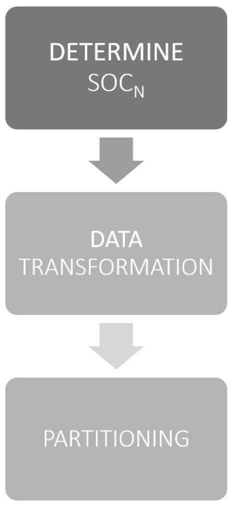

Figure 2.2: Data preparation steps used in this work. Firstly, the normalized SoC is calculated from the existing data. Next step is the data transmormation. Data standardization (normalization) is only applied as transformation technique. Finally, the data is split in multiple smaller sub-sets for the model training.

2.3.1 Determine normalized SoC

The principle of supervised learning is to have an output value also known as target. In SOC estimation, it is the SOC value that must first be determined for each data point. Usually one would use the equation 1.2 or 1.3 for this task. However, as described in chapter 1.2.3, these equations lead to a problem in practice, since a battery is unfortunately affected by temperature and lifetime, and thus the actual SOC range. E.g. the SoC range of *0degC Cycle2* is around 89%, whereas that of *n20degC Cycle4* is only 58%. This means, additional post-processing is required to evaluate this distortion. Instead, the task is assigned to the ANN by using the equation 1.8. According to the readme description of the dataset, the drive cycle tests were terminated for 25°C and 10°C when the voltage first reached 2.5V and for 0°C, -10°C and -20°C after 2.32Ah (80% DOD), 2.03Ah (70% DOD) and 1.74Ah (60% DOD) were discharged. See figure 2.3 for plot. The corresponding code can be found in appendix B.1

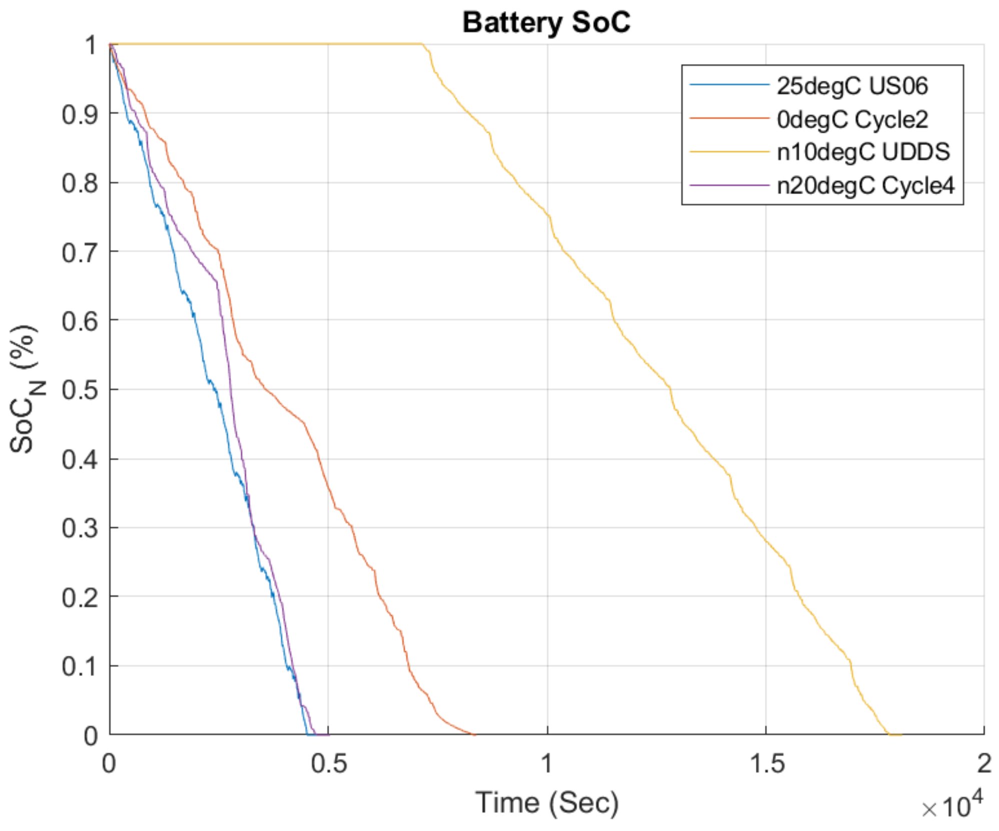

Figure 2.3: Calculated SOC_N with the equation 1.8. All cycles have the same value range. Source: own

2.3.2 Data Transformation

There are different reasons why data needs to be transformed and there are also many transformation techniques for different purposes. Probably the best known is the normalization (or standardization) of input features, so in the domain of SOC estimation. The SOC_N can also be seen as a way of output normalization. Input data normalization is a major step of data preprocessing and is much more common then output normalization. This step rescales the data, so that all features fall within a small specified range, typically to [0, 1], [-1, 1] or other possibilities. Figure 2.4 shows the data spread of voltage, current and temperature of the original Panasonic 18650PF Li-ion dataset.

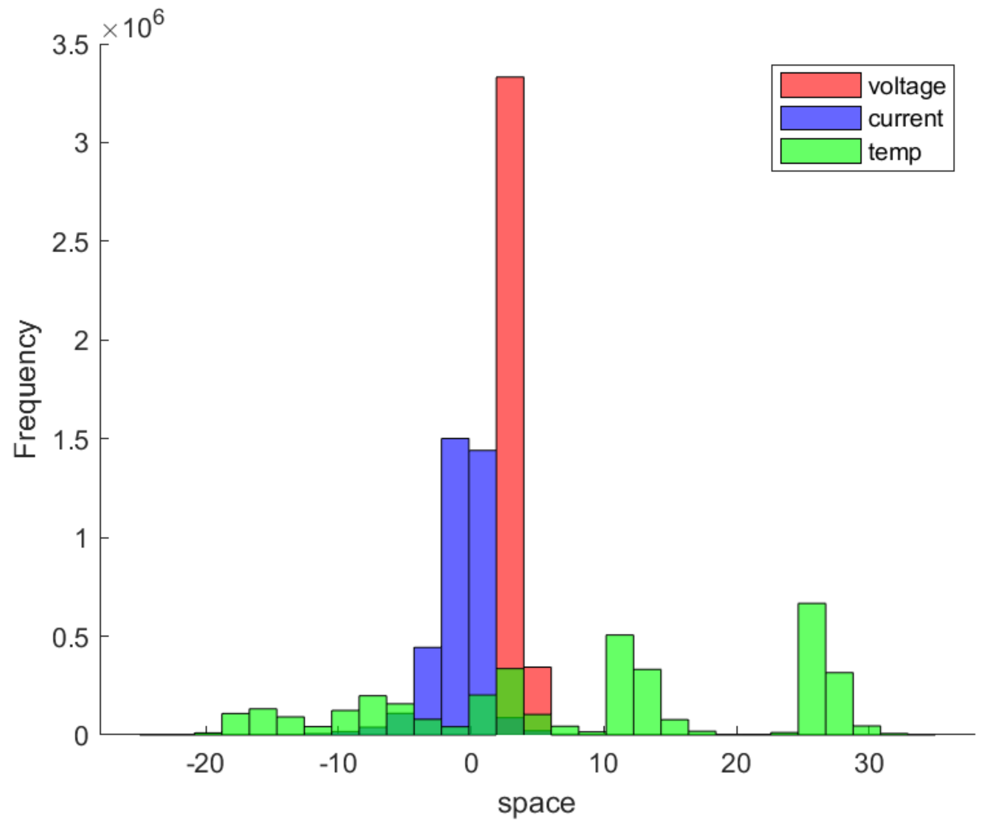

Figure 2.4: Histogram from original dataset. The distribution of voltage, current and temperature is seen here. The data from temperature is far more spread than that of voltage and current

Source: own

Advantages of normalization are the rescaling of features to similar scales, which is a general requirement for many machine learning algorithms. In particular, gradient descent based algorithms will end up having significantly faster updates and better results [SS97, see 11.3.] [BH01]. This is because the feature magnitudes play an important role in weight updates, discussed deeply in 3.1.2. Some weights will get updated faster when compared to others, through the inequality. The mathematical formula for the normalization was described in 1.5. The alternative to Min-Max-Normalization is the z-score standardization, where the data is scaled based on mean (μ) and standard deviation (σ), see equation 2.1.

$$z = \frac{x - \mu}{\sigma} \tag{2.1}$$

These are the most widely used in SOC estimation and both are valid choices. Since the outliers were not handled in this work and Min-Max-Normalization can not handle outliers well, z-score standardization is used. [Dem20, page 71] See figure 2.5. The corresponding code can be found in appendix B.2.

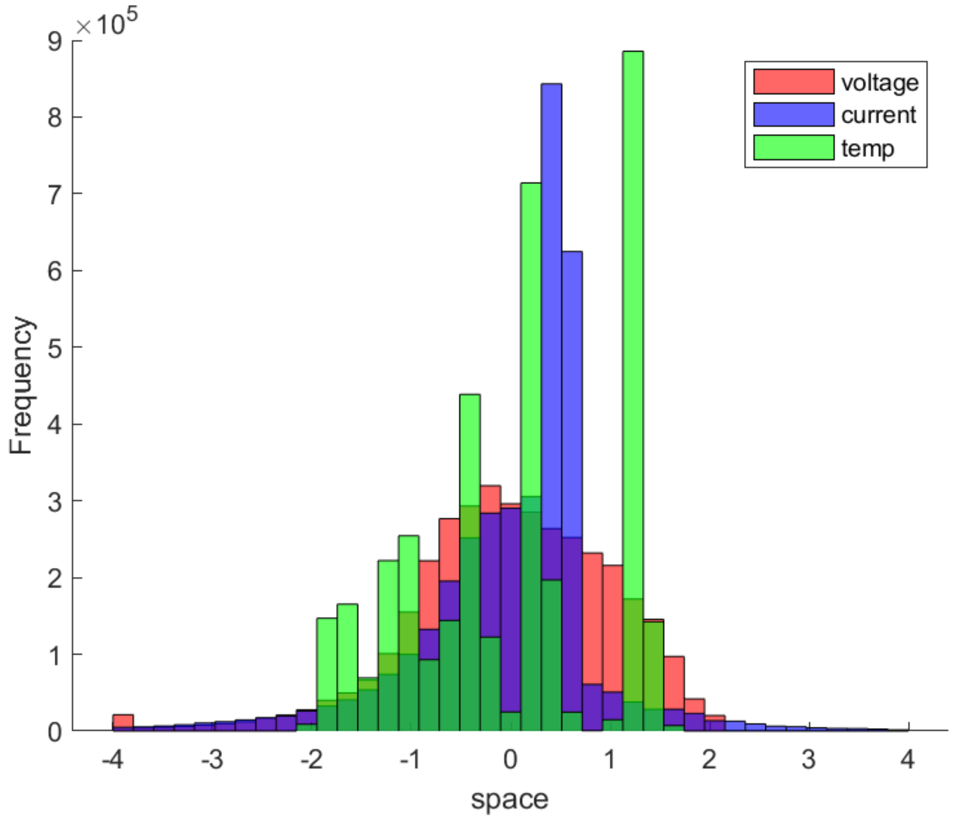

Figure 2.5: Histogram from standardized dataset.

Source: own

There are other transformation techniques in the literatur, which are mentioned here only. Kollmeyer et al. [Che+18] adding gaussian noise to the training dataset to increase SOC estimation accuracy and robustness. Furthermore, Kollmeyer et al. resampling the data in this work [Kol+20] from 10Hz to 1Hz to remove some inconsistency in the data.

2.3.3 Partitioning

The goal of an ML model is to make accurate predictions, beyond those used to train the model. The basis for that is the database. In many model types, it is a requirement to split this data in subsets. [BH01, p. 11.1.] The training subset is the mainly used one to update the weights of the network in the training phase. To evaluate the final predictive performance of a model, a test subset is required. The test subset has to be disjoint from the training subset, to ensure an unbiased evaluation. This is because the network is able to memorize the data from the training subset, therefore the network is hypocritical getting better, but loosing the generalization ability, a phenomenon called overfitting occur, discussed in 3.1.4. The third subset is the validation subset, which is also disjoint from the training and test subset. The validation set is used to unbiased evaluate the ANN during the training phase where the hyperparameters of the model are optimized. Although the network does not learn directly from this set, the architecture of the network adapts to the data. [Rip96, see page 354] Currently, there is no mathematical rule for determining the most appropriate division of the various data sub-sets. But some rules of thumb emerged through experience and trial-and-error. Basheer and Hajmeer (2001) [BH01] have collected some of these rules. For this study, the 65-15-25 (training-validation-test) partitioning is arbitrarily chosen. A random permutation is used to split the dataset, see figure 2.6. For Matlab code, see appendix B.3.

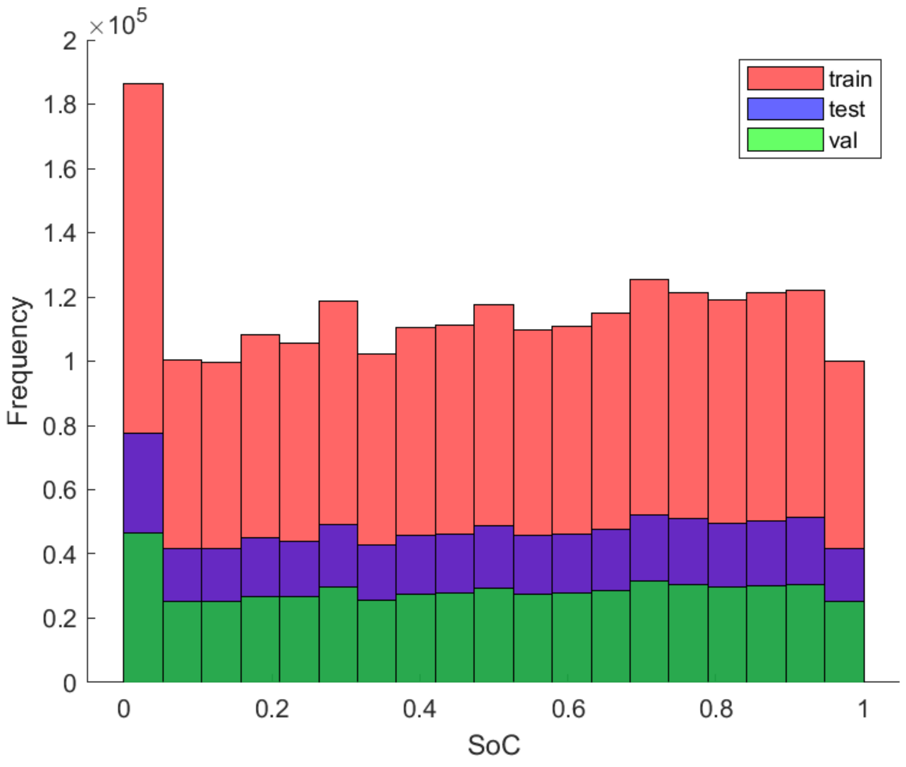

Figure 2.6: Histogram of the dataset after data partitioning. A random permutation is used to split the dataset. This is a 65-15-25 (training- validation-test) split. It is noticeable that there are more values in the range of 0% to 5% than in other ranges.

Source: own

Chapter 3

Design Of The Model

To lay the foundation of this study, this chapter presents technical background and theories for the algorithmic part of this work. All model-relevant decisions for the experimental part of this work are presented and explained in the respective section. The first section 3.1 deals with the topic of the ANNs. For this purpose, the structure of an ANN is explained in a building-up approach - describing it with a mathematical and model representative way. The learning procedure of an ANN requires the use of a complex gradient-based algorithm, which is discussed in section 3.1.2. Followed by two important topics for ANNs - the *activation function* 3.1.3 and *regularization* 3.1.4. The second part of this chapter presents an automated hyperparameter optimizer 3.2 using probabilistic models. The section 3.2.1 gives a form of an overview to the optimization mechanics. The core components of the optimizer are discussed in the following sections 3.2.2 and 3.2.3. Last but not least, an extended mechanic of the optimizer is discussed in section 3.2.4 to overcome constrained optimization problems. The Python implementation can be found in appendix C.

3.1 Artificial Neural Networks

Machine learning (ML) can be seen as the science of programming computers to learn from data. Artificial Neural Network (ANN) is a special type of machine learning. This type is characterized by the fact that artificial neurons are used to build a direct relationship between the features and target, inspired by biological neurons. In the literature, the term unit is often preferred instead of artificial neurons, so this term will also be used in the further course of this work. An ANN is able to learn without task-specific rules by simply seeing data examples submitted by a learning algorithm. The focus of this work is on *(deep) feed-forward neural networks* as mentioned in 1.4. Models are referred to as feed-forward when there are no feedback connections through the outputs of the units back to the inputs or other units. They form the basis of many models. The goal of a feed-forward network is to approximate a function f^*. During the training of a neural network, the

ANN f is directed to f^*. ANNs are called universal function approximators, because of the ability to universally approximate f^*, under certain conditions. [GBC16, chapter 6] The following theoretical explanations refer in particular to regression prediction tasks, that means, functions of the form $f : \mathbb{R}^N \to \mathbb{R}$. [GBC16, see 5.1.1 Regression]

3.1.1 Mathematical and Model Representation of a DNN

The ANN behaviour is stored and distributed among its units' connections. Therefore, learning process of an ANN is nothing more than parameter θ adjustment (Note: Not to be mixed up with hyperparameter). Each unit is constructed by its input $x_{[1xn]}$ and exactly one output y. The figure 3.1 shows the model of a unit and is described further in the course of this chapter.

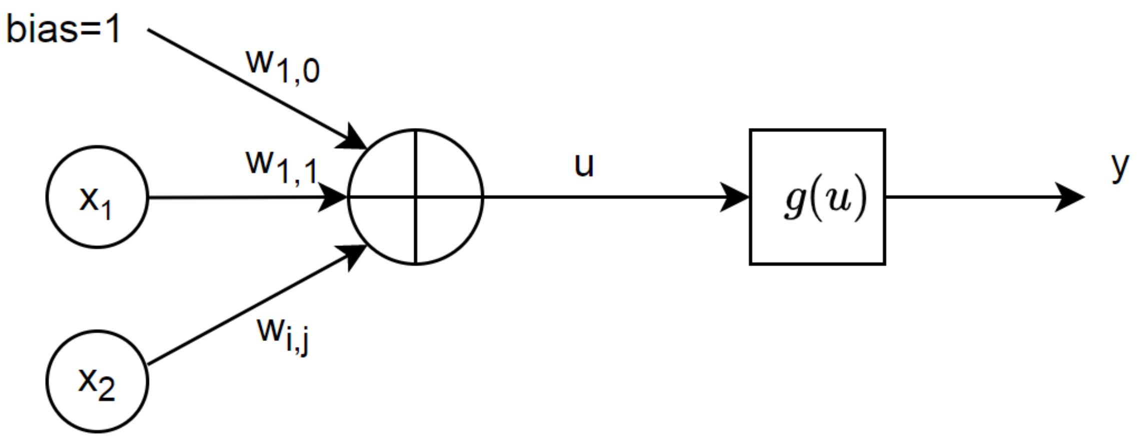

Figure 3.1: A classic artificial neuron model. This neuron has two inputs and a bias which are mapped to the output through the weighted summation and the activation function $g(x)$.

Source: own, inspired by [GBC16, chapter 6]

Formal written, a feed-forward network in its basic form without any hidden layer and with parameter set θ can be described as:

$$y = f(\mathbf{x}; \theta) \tag{3.1}$$

To extend the linearity of this models to the representation for nonlinear functions of x, the linear model is not to apply to x, but to transformed input data $\phi(x)$, where ϕ is a nonlinear transformation. There are multiple possible assignments for ϕ. One is the manual assignment, which was the predominant approach until deep structures emerged. In Deep Learning, the strategy is to learn ϕ by adding additional layer (the so-called hidden layer) to the model. At this way, the model is able to learn $\phi(x)$ automatically. Formally, this can be described as

$$y = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^T \mathbf{w} \tag{3.2}$$

Now the model is able to learn ϕ with parameter θ, and parameter $\mathbf{w}$ to learn the output regression. The most standard is to choose a linear form for the model $f(\mathbf{x}; \theta)$ [Ari+16, equation 6]. Thereby θ is replaced by the parameters $\mathbf{w}$ and $\mathbf{b}$. Note, the axis intersection term is known as bias in many literatures. Formally, this means

$$f(x; W, b) = \mathbf{x}^T \mathbf{W} + \mathbf{b} \tag{3.3}$$

Thus, the layers can be described formally for n layer network as

$$\begin{aligned} \mathbf{h_1} &= f^{(1)}(\mathbf{x}^T; \mathbf{W}_1, \mathbf{b}_1) \\ &\dots \\ \mathbf{h_n} &= f^{(n)}(\mathbf{h}_{n-1}; \mathbf{W}_n, \mathbf{b}_n) \\ y = h_{n+1} &= f^{(n+1)}(\mathbf{h}_n; \mathbf{W}_{n+1}, \mathbf{b}_{n+1}) \end{aligned} \tag{3.4}$$

where $\mathbf{h_i}$ is a vector containing the output of each hidden unit of layer i and $\mathbf{W_i}$ a Matrix containing all unit weights of layer i. $\mathbf{b_i}$ is the vector containing the weights for the axis intersection term. The number of hidden layer and the number of hidden units are hyperparameters, which are going to be optimized.

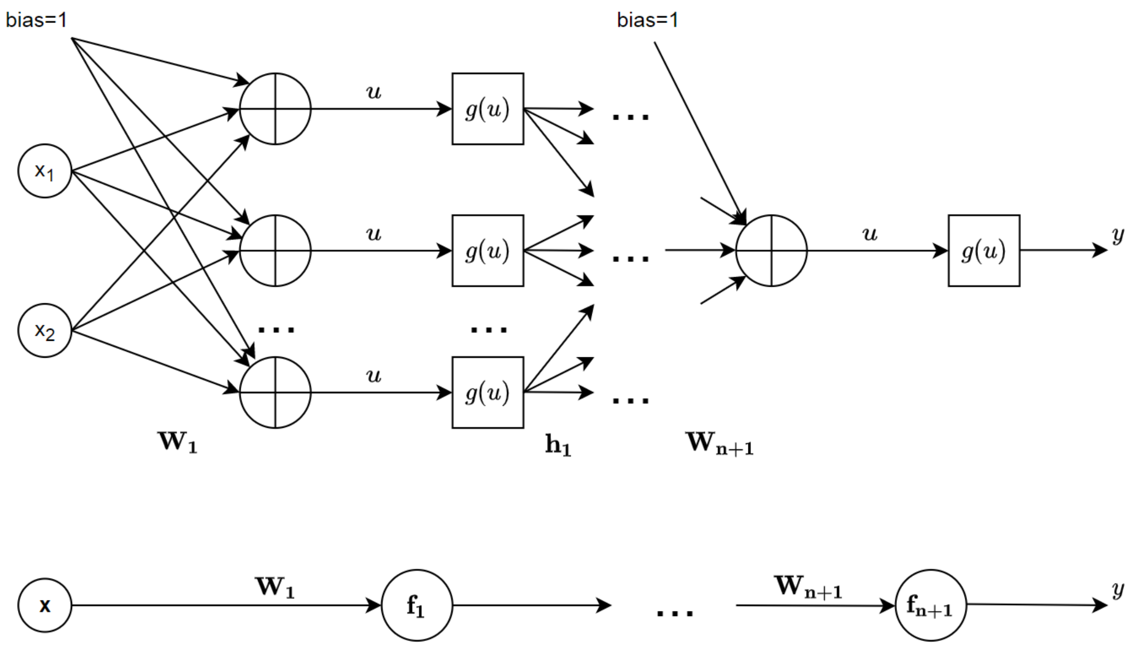

Figure 3.2: (Deep) neural network model. The upper and the lower models are equal. The upper visualize the relationship between the units and layer better. In the lower representation, one node in the graph represents the vector of all units of a layer. It is clearly more compact. The edges in the graph are labelled with: first the parameter matrix $\mathbf{W_i}$ that describes the relationship between two layers and second the scalar values u containing the weighted summation of the input. A matrix $\mathbf{W_i}$ describes the mapping from f_{i-1} to f_i.

Source: own, inspired by [GBC16, chapter 6 and figure 6.2]

The expression 3.4 can be written more compactly. For this purpose, it is pointed out that feed-forward neural networks can be described with a directed acyclic graph, as seen in figure 3.2. This describes, that a function $f^{(i)}$ is composed with all previous functions. This allows to describe the network with only the composition of functions. [GBC16, p. 164] Formally expressed, this means for a network with n hidden layer

$$f(x) = (f^{(n+1)} \circ f^{(n)} \circ ... \circ f^{(1)})(x) \tag{3.5}$$

Applied to the given situation above, the expression can be written as

$$h_i = f^i(f^{i-1}; \mathbf{W_i}, \mathbf{b_i}) \tag{3.6}$$

with $i = 1, 2, 3, ...$ the number of hidden layer and $f^{(0)} = \mathbf{x}^T$. To simplify following expressions further, the intersection term and the weight matrix are concatenated and x supplemented by 1

$$\begin{aligned}\mathbf{W'_i} &= [\mathbf{b_i}, \mathbf{W_i}] \\ \mathbf{x}' &= [1, \mathbf{x}]^T \\ \mathbf{h'}_i &= [1, \mathbf{h}_i]^T\end{aligned} \tag{3.7}$$

The expression 3.6 has strong linear characteristics, since only linear models are used. This can be proven by

$$\begin{aligned}&f^{(i)}(x) = mx + b \\ &f^{(i+1)}(x) = nx + a \\ &f^{(i+1)}(f^{(i)}(x)) = n(mx + b) + a \\ &f^{(i+1)}(f^{(i)}(x)) = nmx + nb + a\end{aligned} \tag{3.8}$$

The result is a linear function. Therefore, to increase the flexibility of the network, every unit getting an *activationfunction* $g(x)$, usually non-linear one. There is a wide range of activation functions, which is why this is discussed in a separate section. [GBC16, p. 169-170] The choice of the activation function is a hyperparameter. Formally, the activation function of a (deep) neural network can be represented as described in 3.9 and shown in figure 3.1 and figure 3.2.

$$h_{i,j} = g(\mathbf{h'}_{i-1}\mathbf{W'}_{i,j}) \tag{3.9}$$

With $\mathbf{h}'_{i=0} = \mathbf{x}'$ and $h_{i=n+1} = f(x) = y$ and j denoting a specific unit in layer i. It makes sense to think of these as single units, since the activation function g is usually applied element-wise. [GBC16, p. 192]

3.1.2 Gradient based optimization

The goal of deep learning is to optimize the error between f and f^*. Optimization in this context mean to minimize/maximize this error by changing x. The most usual way to optimize ANNs is through gradient descent. [GBC16, see 4.3] This work assumes that the reader is already familiar with calculus, but some connections concerning gradients and optimization shall nevertheless be presented here.

If a function $\psi(x)$ has multiple inputs, the change of this function can be checked by the concept of partial derivatives. The partial derivative is written as $\frac{\partial \psi(x)}{\partial x_i}$ and the principle is to check ψ as only the variable x_i increases at point $\mathbf{x}$. In vector calculus, the *gradient* of ψ is the vector of ψ at point $\mathbf{x}$, containing all partial derivatives. The element i of the gradient is the partial derivative of ψ with respect to x_i at point x

$$\nabla_x \psi(\mathbf{x}) = \begin{bmatrix} \frac{\partial \psi(x)}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial \psi(x)}{\partial x_n}(\mathbf{x}) \end{bmatrix} \tag{3.10}$$

[GBC16, p. 82] [6] The gradient has the property of always pointing directly upward to the steepest point. In contrast, the negative gradient $-\nabla_x \psi(\mathbf{x})$ always points downwards. This can be used to optimize ψ and is called *gradient descent*. Formally this means

$$\mathbf{x}^* = \mathbf{x} - \eta \nabla_x \psi(\mathbf{x}) \tag{3.11}$$

where η is called the learning rate, which is a positive scalar with $0 < \eta < 1$ that determines the step size to reduce $\mathbf{x}$ and $\mathbf{x}^*$ is a new point, 'hopefully' deeper than $\mathbf{x}$. [GBC16, p. 83] The learning rate is a hyperparameter.

A function that has to be minimized/maximized is called *objective function*. [GBC16, p. 80] The objective function of an ANN is usually non-convex, which means we have a local optimization problem. That is why an iterative gradient calculation algorithm is used to find a global/local minimum of ψ and the usual used algorithm in ML is known as *backpropagation* algorithm. Mistakenly, the backpropagation algorithm is often understood to be the entire learning algorithm, where it is actually only the gradient calculation. The actual learning process is performed by the optimizer. [GBC16, see 6.5]

The error gradient in an ANN is calculated according to

$$
\begin{aligned}
\nabla_{\mathbf{W}'}\mathbf{J}^i &= \frac{\partial J^i}{\partial \mathbf{W}'_i} && (3.12)\\
&= \dots \\
&= \delta^i \mathbf{y}^{i-1} && (3.13)
\end{aligned}
$$

if the notation of section 3.1.1 is continued. J is the *cost function* and represents the sum of the error by calculating the difference between true and predicted values. There are a variety of cost functions. One of the most widely used is the mean squared error for regression problems. Defined for p training examples as

$$J(\theta) = \frac{1}{p}\sum_{i}^{p}\Big(y - \hat{y})^2 \tag{3.14}$$

where $\hat{y}$ is the labelled output and y the predicted output. [GBC16, see equation 5.4 and chapter 6.1] [ASR20] In practical applications, it is usual to not evaluate the cost function for all data points in the dataset separately, but instead to randomly cumulate data points in minibatches. As a result, the computational effort is significantly reduced based on the batch size. The downside is the increasing standard error over the data, but with a comparatively small factor of hypothetical $\sqrt{o}$, where o is the size of the minibatch. The choice of the batch size depends on computational resources. The trade-off is between convergence time and resulting model performance. Typically, power of 2 batch size ranges are chosen for the minibatch size, especially in use with GPUs. The minibatch size is treated as a hyperparameter. [GBC16, see 8.1.3]

By using the chain rule, the equation 3.12 can be formed to 3.13, [BH01, see 9.1] [GBC16, see 6.5.2, 6.5.3] where δ is a layer depending equation. Formally, δ can be calculated for each unit separate as

$$\delta^i_j = \begin{cases} g'(\mathbf{u}^i_j)(y - \hat{y}) & i = n+1 \text{ (Output layer)} \\ g'(\mathbf{u}^i_j)(\sum_k \delta^{i+1}_k \mathbf{W}'^{i+1}_k) & i \leq n \text{ (Hidden layer)} \end{cases} \tag{3.15}$$

where j denote the unit in layer i and k the connected unit in layer $i + 1$. $g'(u)$ is the derivate of the activation function $g(u)$. Because of the interdependence of the layers, the values of layer i influence all elements of layer $i + 1$, which is why the case $i \leq n$ is more complicated.

At the beginning of this section, a gradient descent algorithm was presented, to find the optimal x. Unlike the backpropagation algorithm, it contains a parameter update equation 3.11. This equation is called *stochastic gradient descent* (SGD) and there are a wide range of optimization algorithms. [GBC16, see 8.3.1 - 8.5.3] SGD is the oldest method to optimize the parameters of an ANN, which is still in use and the other optimizers can be seen as SGD updates. [GBC16, see 8.3] Unfortunately, there is currently no mathematical way to find the best optimizer for a specific model or data. [GBC16, see 8.5.4] There exist different sources, where multiple optimizers are compared, but the results vary from research to research. Therefore, the comparison of [KB17] is used as a basis, which moreover introduce the *Adam* '*adaptive moment*' optimizer. The results show, that Adam is indeed a good choice and many experts are also convinced about Adam, which is why the Adam optimizer is used for the parameter optimization task. The algorithm of Adam can be seen in algorithm 1

Algorithm 1 The Adam-Algorithm

```
Require: $\eta$ : Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates (Recommended defaults: 0,9 and 0,999)
Require: $\epsilon$ : Constant for numerical stabilization (Recommended default: $10^{-8}$)
Require: $\theta$ : Initial parameter vector
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
  while stopping criterion not met do
      Sample a minibatch of size $m$ : $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$
      $t \leftarrow t + 1$
      $g_t \leftarrow \frac{1}{m} \nabla_\theta \sum_i L(f(x^{(i)}; \theta), y^{(i)})$ (Calculate gradient, see expressions 3.12 - 3.15)
      $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$ (Update biased first moment estimate)
      $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ (Update biased second moment estimate)
      $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$ (Correct bias in first moment estimate)
      $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$ (Correct bias in second moment estimate)
      $\theta_t \leftarrow \theta_{t-1} - \frac{\eta \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$ (Update parameters)
  end while
```

3.1.3 Activation functions

The property of ANNs is to approximate all possible functions. By using activation functions $g(x)$, a model is able to better fulfil this property. An important prerequisite for activation function is the differentiability of the function, since it is required for the delta rule 3.15. The choice of activation functions can be considered as hyperparameter. At this point, it must be mentioned that the selection of the activation functions has not been dealt with in detail in this work. Instead, the selection task has been assigned to the hyperparameter optimizer; discussed in section 3.2. Some popular regression activation functions are presented below, which are used in this work. An extensive discussion and list of activation functions can be found in Nwangpa et al. [Nwa+18].

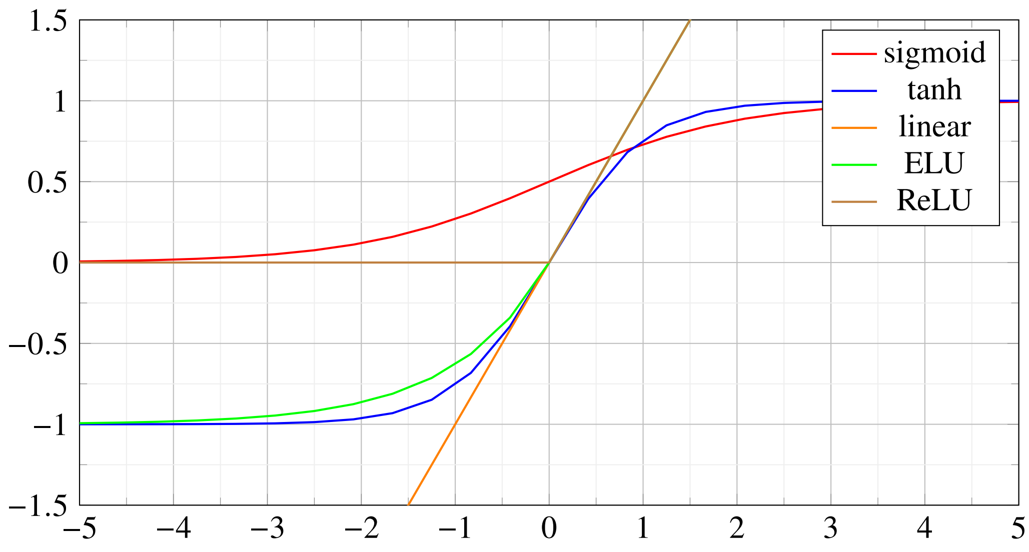

Figure 3.3: Graph of different used activation functions. It depicts sigmoid, tanh, linear and ELU activation function

Source: own

Linear

The linear activation function is a straight line function where activation is proportional to the parametrized input. It is defined as

$$\begin{aligned} g(x) &= ax + b \\ g'(x) &= a \end{aligned} \tag{3.16}$$

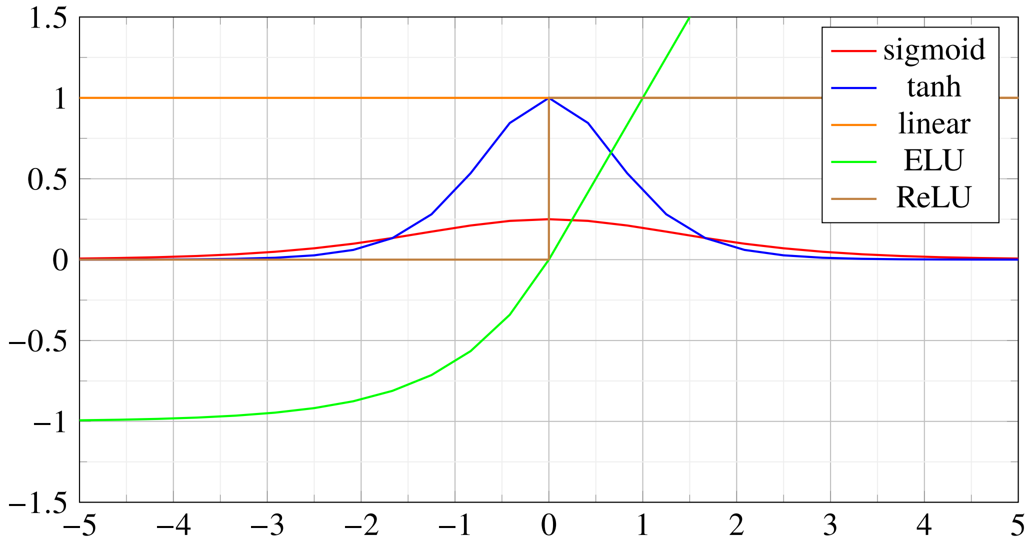

Figure 3.4: Graph of different derivatives of used activation functions. It depicts the derivative of sigmoid, tanh, linear and ELU activation function
Source: own

The linear activation function is a common choice for regression problems, since the function value range is not restricted. However, as one can see, the derivative is a constant, which means that the gradient in the backpropagation has no relationship with x. This activation function is only relevant for the output unit, since it allows an even distribution of the value ranges. The usual choice for the constants are $a = 1$ and $b = 0$, which is equivalent to no activation function.

Sigmoid

The Sigmoid is a monotonic function that takes a real value as input, and outputs a value between 0 and 1. A major advantage of the Sigmoid activation function is its nonlinearity, which allows it to represent nonlinear functions well. It is defined as

$$\begin{aligned} g(x) &= \frac{1}{1+e^{-x}} \\ g'(x) &= g(x)(1-g(x)) \end{aligned} \tag{3.17}$$

However, the Sigmoid suffers from gradient saturation. As figure 3.4 shows, the derivative of sigmoid goes close to zero for high positive or negative values, and thus that of the gradient 3.15; also known as *vanishing gradient problem*. A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session, especially in deep networks is this a major problem. The output of the Sigmoid is not zero centred, which makes optimizing harder. [Nwa+18, see Sigmoid] Values close to zero or one are hard to reach, which is why sigmoid is not used on output unit.

Hyperbolic Tangent Function

The hyperbolic tangent function (tanh) is in principle only a scaled and shifted sigmoid function. However, the function is point symmetric to the origin and has the value range -1 < g(x) < 1, so it also allows negative outputs. Therefore, negative effects on the values are possible. The tanh activation function is defined as

$$\begin{aligned} g(x) &= \frac{e^x - e^{-x}}{e^x + e^{-x}} \\ g'(x) &= 1 - tanh^2(x) \end{aligned} \tag{3.18}$$

The tanh suffers from vanishing gradient as well.

Partly Linear activation functions

The partial linear activation functions (so designated in this paper) are a group of activation functions that are variations of the Rectified Linear Unit (ReLU) activation function and are linear for $x < 1$. ReLU is defined as in 3.19.

$$\begin{aligned} g(x) &= \begin{cases} x & x > 0 \\ 0 & x <= 0 \end{cases} \\ g'(x) &= \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases} \end{aligned} \tag{3.19}$$

The exponential linear unit (ELU) is a partly linear function defined as

$$\begin{aligned} g(x) &= \begin{cases} x & x > 0 \\ a(e^x - 1) & x <= 0 \end{cases} \\ g'(x) &= \begin{cases} 1 & x > 0 \\ ae^x & x < 0 \end{cases} \end{aligned} \tag{3.20}$$

These functions differ for values $x < 0$. All of the partly linear functions overcome the vanishing gradient problem, which is why they have become very popular and are the current state-of-the-art. Compared to ReLU, ELU also has negative values, allowing to push the mean of the unit activation closer to zero, which leads to a faster network training. Overall, ELU has a faster convergence than ReLU or other variations. [CUH16] a is a constant and contains by default the value 1. This can be treated as a hyperparameter.

3.1.4 Regularization

One of the main aspects of ML is to train models that not only have sufficient prediction results on the training data seen during the training process, but also can interpolate unseen data with similar error in the same problem domain. [GBC16, chapter 7] This is called the *generalization* of an ANN model. A model usually generalize better, the more training data is available. In ML, the most used terminologies used when talked about the generalization of new data are called *underfitting* and *overfitting*. An underfitted ML model is not a suitable model and will have poor performance on the training data and test data. Overfitting occurs when the distance between training error and test error is too large. This happens, when a model focuses too much on reducing training errors; it interpolates the noise in the training data as well, that occurred randomly. [Sri+14, see Introduction] [GBC16, section 5.2 and figure 5.5] A validation dataset is usually used to measure the generalization of the model. The test error can be calculated in the same way as the training error 3.14. Regularization procedures aim to increase the generalization ability of ML models. In doing so, many methods in deep learning are explicitly designed to reduce test error, but this can lead to higher training error. [GBC16, chapter 7] Two regularization procedures are presented below, which are used in this work.

Dropout

Dropout selects some nodes with probability p in the network and removes them along with all their incoming and outgoing connections as shown in figure 3.5. The dropout is reapplied for every iteration. The random drop can be interpreted as a way of regularizing a neural network by adding random noise to its hidden units. [Sri+14] This can be formally written as

$$\begin{aligned} r_{i,j} &\sim Bernoulli(p) \\ \tilde{\mathbf{h}}_i &= \mathbf{r}_i \mathbf{h}_i \\ h_{i+1,j} &= g(\tilde{\mathbf{h}}_i' \mathbf{W}_{i+1,j}') \end{aligned} \tag{3.21}$$

with layer i and unit j of i. p is the probability of retaining a unit, and is called *dropout rate* with a value range of $0 < p < 1$. $p = 1$ implies no dropout and $p = 0$ the dropout of all units. Note that *Keras* implementing $p = 1$ as full dropout and $p = 0$ as no dropout. [7] The dropout rate is a hyperparameter. The equivalent model is shown in figure 3.5.

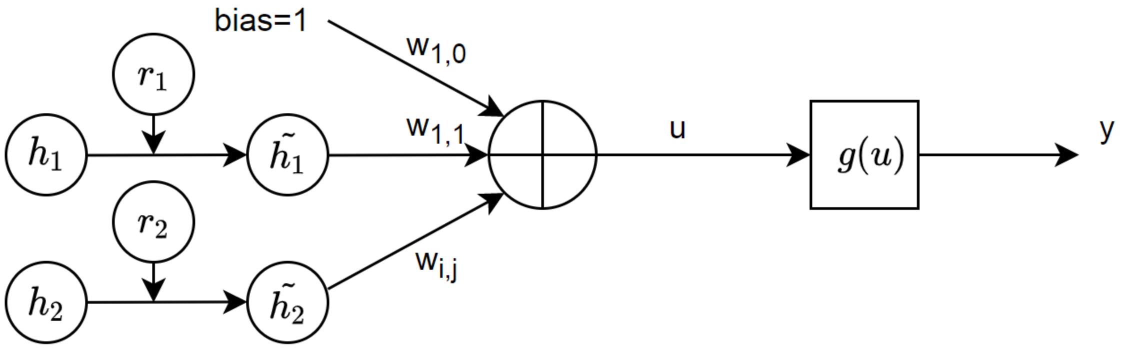

Figure 3.5: A dropout model of a neural unit. The variable r_i decide if a unit is retain or not. r_i is a Bernoulli distributed variable with a probability p.

Source: own, inspired by [Sri+14, Figure 3]

Dropout is little computational expensive but easy to use and usually produces very good results, and therefore, it is most probably the most frequently used regularization technique in the field of deep learning. Furthermore, the dropout is only used during the training and the resulting neural network is used without dropout. At test time, the weights are scaled as $\mathbf{W}_i = p\mathbf{W}_i$. Dropout cannot be used in the Input or Output Layer because the model would break down. [Sri+14] [GBC16, see 7.12]

Early Stopping

Every model reach its optimal prediction capacities at some point during training. Usually, an overfitting is observed afterwards, or ideally the prediction abilities remain nearly constant and costs unnecessary computational effort. An intuitive approach for the solution is to stop the parameter θ setting at an early stage of the training. This approach is the idea behind early stopping. Concretely, one is looking for the state where the validation error is lowest. Each time, a better validation error is found, the parameter set has to be saved. And if there can not be detected an improvement over a predefined number of iterations, the training algorithm terminates. This prevents unnecessary further training. On the other hand, there is the risk to land on a local minimum on the objective function and to never reach the global one, since the evolution of the validation error is not smooth in many practical cases, which can be seen in figure 4.3. Therefore, early stopping is not always advisable. The trade-off is between generalization and training time of the model. Due to the very limited computational resources available 2.1.1, early stopping is used in this work. However, the number of predefined iterations is chosen largely. [GBC16, see 7.8]

3.2 Bayesian Optimization Of Hyperparameter

(Note: The mathematical assignments are redefined for this section)

Most machine learning algorithms have setting parameters that must be specified outside the learning algorithm itself, called hyperparameters. The correct choice of hyperparameters is of utmost importance for the precise prediction of the model. Some hyperparameters were mentioned in section 3.1.1. The aim can be formally described as

$$\arg\min_{x \in X} f(x) \tag{3.22}$$

where $f : X \rightarrow \mathbb{R}$ is the objective function to minimize, such as an error metric, on the validation data for an ANN. x is a set of hyperparameters in the domain X.

Hyperparameter determination poses a problem, since the evolution of the objective function of an ANN is typically non-convex, which means there can be multiple local minima and there is not a practical way to say which is the global minimum. If one is an expert, the hyperparameters can be adjusted manually by hand tuning to fulfil the expression 3.22. Other methods are based on exhaustively searching the hyperparameter space X. For example, the grid search technique searching the best set of hyperparameter in a grid of discretized spaced values of X. Grid search can be perfectly parallelized, since each evaluation is independent, but this technique is extreme expensive to evaluate with complexity of n^m, where m representing the number of hyperparameters and n

the number of values per hyperparameter. Random search is a technique where the systematical evaluation of every combination is replaced by a simple random evaluation of the space. However, finding the minimum of X strongly depend on its dimension, similar to grid search. [BYC13][8] These tuning methods are difficult for many models, since the evaluation of a single hyperparameter set is computationally very expensive. Therefore, techniques using probabilistic approaches are used to reduce the number of evaluations.

Bayesian optimization (BO) is a global optimization method, i.e. a method with which it is possible to determine or approximate the global optimum of a black-box objective function. Black box in this case refers to the fact that the behaviour of f is not known. BO is a very effective algorithm, regarding the number of evaluations needed for sufficient results, and therefore, it converges to a near optimal configuration in few iteration steps. It is used especially where the optimization algorithm needs to be derivative-free, and is also applicable when the function is non-convex. [Fra18, see Introduction][MVH21] However, BO also has its limitations for the estimation of SOC_N. In theory, BO can optimize an unlimited finite number of hyperparameters, but in practice, as a rule of thumb, the number is limited to about 20, and the fewer the better. Furthermore, the observed $f(x)$ should be as noise-free as possible, since the convergence is slowed down. [MVH21, see II][Fra18]

3.2.1 Bayesian optimization

The main idea of BO is to use the information of all previous evaluated points to make a new probabilistic estimated evaluation. To generate and update this probabilistic model, BO using the Bayes' theorem, see theorem 1

Theorem 1 *The a posteriori probability of a model M given data D is proportional to the likelihood of D given M multiplied by the a priori probability of M:*

$$P(M \mid D) \propto P(D \mid M)P(M) \tag{3.23}$$

[BCF10, page 2] Note, this is the adapted Bayes' theorem.

In Bayesian optimization, the *a priori* probability describes the prior knowledge about the space of possible objective functions. For example, if the assumption is that the objective function tends to be smooth, then the objective functions with high variance should have a low *a priori* probability. The *a posteriori* probability correspondingly describes the updated probabilistic model of the objective function, which is also sometimes called a *surrogate function*. The surrogate function gives an estimation of the objective function. [BCF10, page 2-3] Besides the probabilistic model, another crucial aspect of BO is the use of a so-called *acquisition function* to determine the next evaluation point. Importantly, the acquisition function is responsible for the trade-off between uncertain search regions called exploration and high expected search regions called exploitation. This behaviour leads to effective sampling of the search space. A determination of the maximum of this function selects the next data point to be used. [BCF10, page 3] The concept of BO can be seen in figure 3.6. So to perform Bayesian optimization, there are two important topics to consider. First, the way of modelling the *a priori* probability, for which usually Gaussian processes are used [BCF10, section 2.1], and second, the choice of the acquisition function.

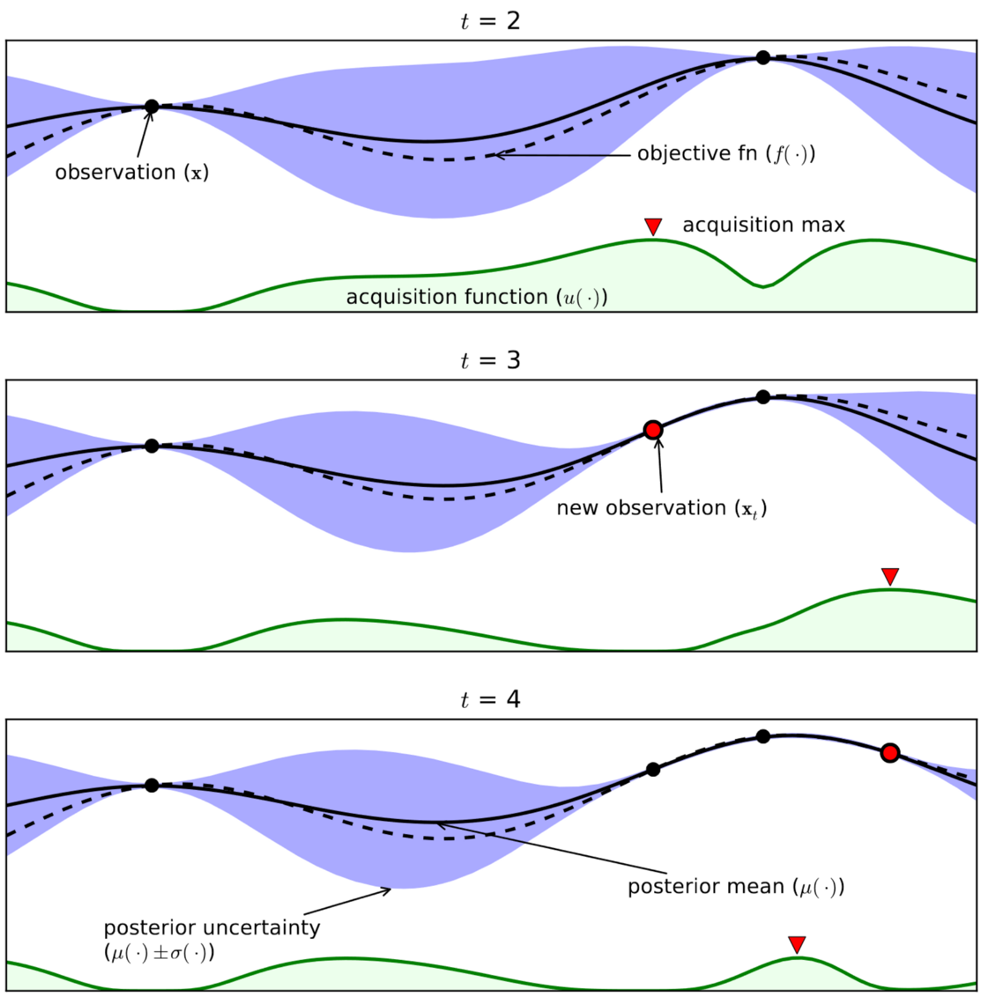

Figure 3.6: Bayesian optimization principle on 1 dimensional problem. The dashed line is the objective function. The solid line is the mean of the Gaussian process constructed from the observations. The blue area is the mean ± variance, which shows the uncertainty. At time t=3, a new observation is added to the *a priori* model and the model is updated. This sampling point was obtained from the maximum of the acquisition function (green) at time t=2. The acquisition function is high where the Gaussian process predicts a high objective and where the prediction uncertainty is high. This behaviour ensures the exploration-exploitation trade-off.

Source: [BCF10, figure 1]

3.2.2 Gaussian Process

A Gaussian process (GP) is a case of a stochastic process (a collection of random variables indexed by time or space) in which every finite collection of random variables are multidimensionally Gaussian distributed, i.e. any finite linear combination of them is normally distributed. Thus, Gaussian processes are a natural generalization of the multivariate Gaussian distribution. Analogous to a multivariate Gaussian distribution, which is completely and unambiguously determined by the expected value and the covariance matrix, a Gaussian process is completely and unambiguously determined by an expected value or mean function $\mu(x)$ and a covariance function $k(x, x')$. This is formally described with

$$f(x) \sim \mathcal{GP}\big(\mu(x),\ k(x, x')\big) \tag{3.24}$$

There are several options to estimate the prior mean function. Ideally, one would like to have $\mu(x) \approx f(x)$. If no prior knowledge about $f(x)$ is available, constant zero $\mu(x) = 0$ is an adequate option. Otherwise, the estimation can be done over historical data. The used BO package offers the estimation via additional data points, either by random or fixed chosen points. [MVH21, see II.B] [BCF10, p. 8] More interestingly is the covariance function, which is the part, relating one observation to another; described further below in this section.

Given a set of noisy observations $D_{1:t} = \{\{x_1, \tilde{f}(x_1)\}, ..., \{x_t, \tilde{f}(x_t)\}\}$ with a sampling noise $\epsilon \sim \mathcal{N}(0, \sigma^2_{noise})$ so that

$$\tilde{f}(x_i) = f(x_i) + \epsilon_i \tag{3.25}$$

applies and an unknown next point x_{t+1}, the Gaussian process is computed as

$$P(f(x_{t+1}) \mid D_{1:t}, x_{t+1}) = \mathcal{N}\big(\mu_t(x_{t+1}), \sigma^2_t(x_{t+1}) + \sigma^2_{noise})\big) \tag{3.26}$$

with

$$\begin{aligned}
&\mu_t(x) = \mathbf{k}^T \mathbf{K}^{-1} \mathbf{f}(x_{1:t}) \\
&\sigma^2_t(x_{t+1}) = k(x_{t+1}, x_{t+1}) - \mathbf{k}^T \mathbf{K}^{-1} \mathbf{k} \\
&\mathbf{k} = \begin{bmatrix} k(x_{t+1}, x_1) & k(x_{t+1}, x_2) & \cdots & k(x_{t+1}, x_3) \end{bmatrix} \\
&\mathbf{K} = \begin{bmatrix} k(x_1, x_1) & \cdots & k(x_1, x_t) \\ \vdots & \ddots & \vdots \\ k(x_t, x_1) & \cdots & k(x_t, x_t) \end{bmatrix} + \sigma^2_{noise} I
\end{aligned} \tag{3.27}$$

[BCF10, see 2.1 and 2.4] Therefore, to generate a Gaussian process that can be used as an a priori probability, it is necessary to define an expected value function and a covariance function. The choice of the covariance function is an important part of the GP. In conjunction with the covariance matrix, it describes the shape of the distribution, and ultimately determines the characteristics of the GP. There are many predefined covariance functions, but one can also create new ones by combining them in different ways, e.g. in *Maggi et al.* [MVH21]. A commonly used predefined kernel is the Matérn 5/2 kernel, which is also used in this work. The comparison in *Snoek et al.* [SLA12, see fig.5] was used to make the decision, and this is the default kernel of the used BO package. The general Matérn kernel is defined as

$$k(x,x') = \frac{1}{2^{\varsigma-1}\Gamma(\varsigma)}\Big(\sqrt{2\varsigma}\, d_\ell(x,x')\Big)^{\varsigma} H_\varsigma\Big(\sqrt{2\varsigma}\, d_\ell(x,x')\Big) \tag{3.28}$$

where $\Gamma(.)$ is the Gamma function, $H_\varsigma(.)$ is the modified Bessel function and $d_\ell(x,x') = \sqrt{\sum_k (x_k - x'_k)^2/\ell_k^2}$. ς represents an additional parameter that can be interpreted as a general smoothness parameter. [MVH21, see equation 6] Usual choices for the parameter are $\varsigma = 1/2$ (once differentiable) or $\varsigma = 5/2$ (twice differentiable). Some implementations of the Matérn kernel include an additional hyperparameter called *length scale* ℓ parameter, which influences the strength between the x's. There are various ways to optimize the kernel hyperparameter. However, there is no information about the implemented method in the used BO package. Some methods can be found in Frazier et al. [Fra18, see 3.2]. [BCF10, see 2.2]

3.2.3 Acquisition Function

An acquisition function is a mathematical technique that control how the parameter space should be explored during BO. Learning procedure optimization with hyperparameters usually aims to minimize loss-values. Therefore, the acquisition function should have high values at points where a low value of the objective function is very likely. However, acquisition functions and the BO algorithm is usually the opposite and is defined to solve maximization problems. Instead of changing the BO algorithm, one can change a minimization problem to a maximization problem just by a simple transformation of the objective function with

$$f(x) = -f(x) \tag{3.29}$$

[BCF10, p. 4] The acquisition function is the component of the BO, primally responsible for the exploration and exploitation trade-off in the selection of additional points. This behaviour is in-built in the nature of acquisition functions. The *expected improvement* (EI) is a very common choice as an acquisition function. [MVH21, see IV] EI is defined as

$$\begin{aligned} EI(x) &= \begin{cases} \big(\mu(x) - f(x^+) - \xi\big)\Phi(Z) + \sigma(x)\phi(Z) & \text{if } \sigma(x) > 0 \\ 0 & \text{if } \sigma(x) = 0 \end{cases} \\ Z &= \frac{\mu(x) - f(x^+) - \xi}{\sigma(x)} \end{aligned} \tag{3.30}$$

where Φ and ϕ are the cumulative and probability density function of the standard normal distribution. $f(x^+)$ is the best so far evaluated point, formally $f(x^+) = max_{i=0,\dots,n} f(x_i)$. ξ is a parameter to control the trade-off strength between exploration and exploitation. A general well working choice is $\xi = 0.01$, which is also defaulted in the used BO package. The EI acquisition function has a particularly desirable behaviour during optimizations. [BCF10, see 2.3.1-2.3.2] To determine the maximum of the acquisition function, there are several efficient methods. The BO package default is the *Limited-memory Broyden–Fletcher–Goldfarb–Shanno* (L-BFGS) algorithm, which is a first derivative quasi-Newton method, which is a common choice. [MVH21, see IV]

3.2.4 Constrained Optimization

Sometimes a problem is valid or well-defined only for points in a certain region. In case of BO, this means the space of possible candidates of x are not freely explorable. An inequality constrained space can be described formally as

$$\begin{aligned} x^* = \arg\min_{x \in X} & \tilde{f}(x) \\ & \text{subject to} \\ & c_1(x) \leq \lambda_1 \\ & \dots \\ & c_p(x) \leq \lambda_p \end{aligned} \tag{3.31}$$

with the noisy objective function $\tilde{f}$ as described in 3.2.2 and the constraints $c_k : X \longrightarrow \mathbb{R}$. The constraints are known and describing the feasible region of the problem. BO is a powerful tool to deal with those multi inequality constrained problems. There are several approaches to solve this type of problems by changing the acquisition function itself. Popular approaches include to modify the *expected improvement*, which is also used from the used BO package. The exact implemented methods are not known, however the authors of the package show here [9], that the package can solve constrained problems for $\lambda = 0$. Some approaches for constrained BO can be found here [Sha+16, VIII.A] [Gar+14, see in particular 3.1-3.2] [UB21].

In this work, constrained BO is used to determine the number of hidden layer of the network. Given the number of units $x_i \in \mathbb{N}_0$ per layer i , the constraints are set as

$$\begin{aligned} x_i &\leq x_i \cdot x_{(i-1)} \\ x_i - (x_i \cdot x_{(i-1)}) &\leq 0 \end{aligned} \tag{3.32}$$

Thus, if $x_{i-1} = 0$, then according to the inequality the following layer x_i is only for 0 true. This ensures that no subsequent layer x_i can exist if there is no layer x_{i-1}.

Chapter 4

Experiment And Results

In this chapter, the results of the experiment are presented and interpreted. The previously discussed topics are applied here. Section 4.1 shows the setup of the experiment, including the hyperparameter search spaces. In section 4.2 the used performance metrics are discussed, which helps later in the analytical part to express the model performance. In section 4.3 and section 4.4 the results of the experiment are discussed. Furthermore, the shortcomings of the experiment are addressed in these sections. The Python code for the experiments can be found in appendix C or online at `https://github.com/KocFatih/Thesis_BS`.

4.1 Experimental Setup

The scheme in figure 4.1 illustrates the working steps of the Bayesian optimizer, designing the ANN model described in the previous chapters.

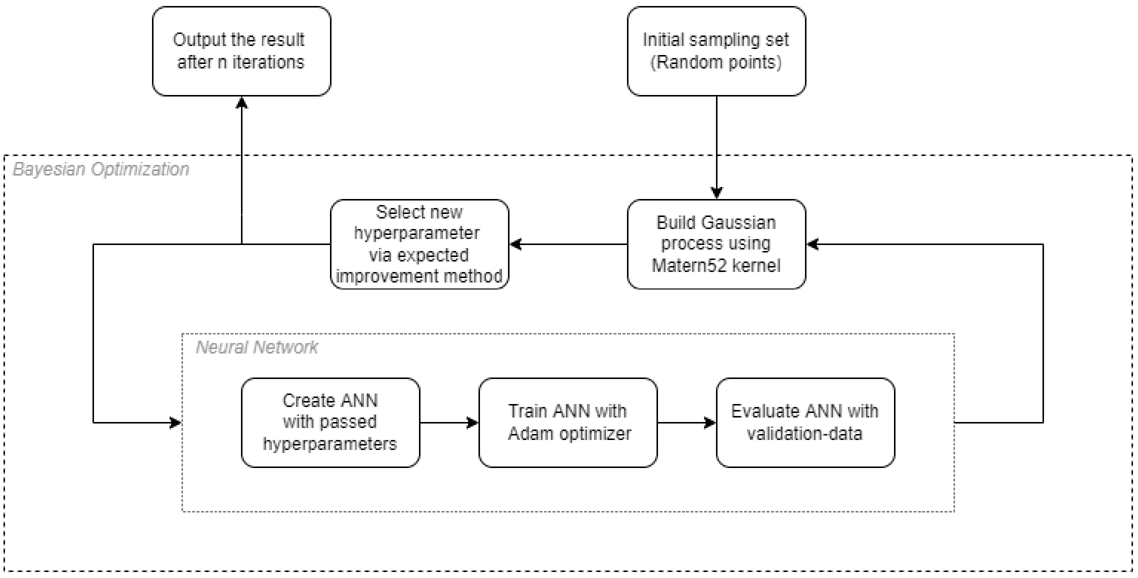
Output the result after n iterations
Initial sampling set (Random points)
Bayesian Optimization
Select new hyperparameter via expected improvement method
Build Gaussian process using Matern52 kernel
Neural Network
Create ANN with passed hyperparameters
Train ANN with Adam optimizer
Evaluate ANN with validation-data

Figure 4.1: The scheme of the ANN and BO experiment is shown here. The BO is getting initialized first with random points (or alternatively with fixed points). The optimizer sequentially tests the ANN model with newly selected hyperparameter sets and updates the surrogate function after the network evaluation is complete.

The aim is to determine the optimal hyperparameter set that leads to the most accurate SOC_N estimation possible. The prepared data is used, which contains five ambient temperature ranges with nine driving cycles each, to cover a wide range of applications. The hyperparameter search space can be found in table 4.1.

Table 4.1: The table shows the hyperparameter search space

Hyperparameter	**Search Space**
Num Hidden Nodes 1	3 to 50
Num Hidden Nodes 2	0 to 50
Num Hidden Nodes 3	0 to 50
Num Hidden Nodes 4	0 to 50
Num Hidden Nodes 5	0 to 50
Learning Rate	$1e^{-6}$ to $1e^{-2}$
Activation Hidden	tanh, Sigmoid, ELU
Activation Output	linear, ReLU
Dropout	0.0 to 0.9
Batch Size	64, 128, 256, 516, 1024

Note that the boundaries are chosen sparse due to the limited computational resources, see 2.1. In particular, the number of units and the number of hidden layers strongly impact training time, since the number of the adjustable parameters in a fully connected network increases drastically. It's important to choose a wide range for the learning rate, as the Adam optimizer's default setting often leads to unstable training in terms of overfitting and strong fluctuations in the learning process. A batch size of 32 should also be considered, but this leads to long training times due to the many parameter updates, explained in 3.1.2. Additionally to *Early Stopping* 3.1.4, 200 iterations of the training data are used as stopping criterion of the model, which is a sparse choice and should choose larger. The budge of optimization iterations for the BO algorithm is set to 110 in total, consisting of 15 initial random points and 95 optimization iterations. With the hardware and software mentioned and the above hyperparameter space, the average training for a specific hyperparameter set configuration take about 2 hours, ranging from 30 minutes to 6.5 hours. The total optimization time take about 9 $\frac{1}{3}$ days.

4.2 Used Performance Metrics

The performance of a regression model for supervised learning can be expressed as the mean error between model predictions and the labelled values. The obtained value is a scalar, which indicates the capabilities of the model. The three most common metrics choices are discussed below. All three are recorded in this work, but the MAE is the mainly used one for the BO algorithm and the evaluations of the experiment. A separate test set is used to evaluate the performance of the model in an unbiased way, see 2.3.3. This allows to detect overfitting of the model 3.1.4.

Mean Absolute Error

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i| \tag{4.1}$$

The Mean Absolute Error (MAE) describes the mean absolute deviation of the predictions y from the expected values $\hat{y}$. The unit of the MAE is the same as the original unit of the target value, which is easy for a human to interpret. The MAE is an intuitive metric, since the changes are linear.

Mean Squared Error

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{4.2}$$

The Mean Squared Error (MSE) describes the mean square deviation between predictions y and expected values $\hat{y}$. The MSE has a quadratic effect in its value changes. The error unit is not the same as the target.

Root Mean Squared Error

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{4.3}$$

Root Mean Squared Error (RMSE) is an extension of MSE. The root of MSE is taken to set the unit in relation to the target.

4.3 Bayesian Optimization Results

The right (blue) plot of the Figure 4.2 shows the progress of the optimization recorded during the Bayesian optimization process, displayed with the MAE metric. The left (red) plot visualize how the algorithm explored the search space by looking at the distance between consecutive evaluations.

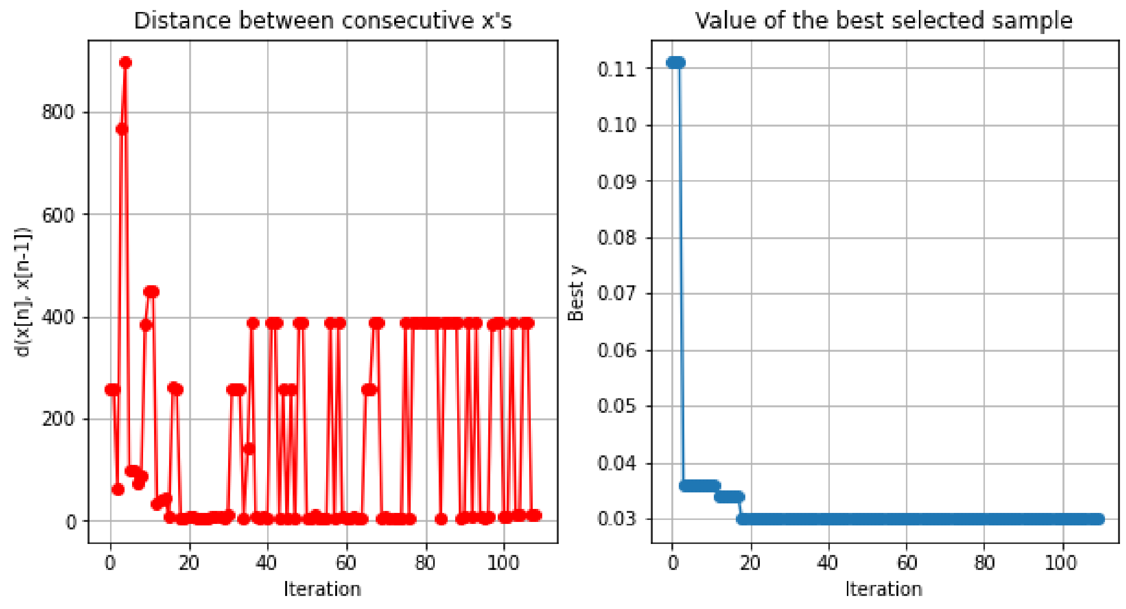

Figure 4.2: The figure shows the results of the BO experiment. The left plot shows the distance between consecutive hyperparameter sets. The tradeoff between *exploration* and *exploitation* is visible. The right plot shows the MAE of the best hyperparameter set found so far. The best result is found at iteration 73, which can not be seen directly from the figure.

The left plot shows the tradeoff between *exploration* and *exploitation*. Lower values mean that the optimizer is trying to find a high value in a specific region of the search space, whereas a high value indicates a global exploration in other regions of the space. It can be seen that the distances of the first 15 iterations are rather random due to the random initialized samples. Thenceforth, the left plot appears to fluctuate constantly between around 0, 260, and 390 units. Occasionally, there are consecutive evaluations that are very close to each other. These evaluations usually correspond to a decrease in the value of the best found sample, or the value is close to the minimum. The long search after iteration 19 can also be explained by this. Here it is particularly long because a very good value was found shortly after the random initialization, which has a high impact on the *acquisition function*. Note that the best value found is at iteration 73, which is not directly visible from the right plot. The optimizer was in iteration 19 able to find a value, which is close to the best found value. The detailed data of the evaluations sorted by performance in descending order can be read from the table D.2 or alternatively the unsorted data can be found in appendix D.1.

4.4 Artificial Neural Network Results

The best found architecture for the model is a DNN with five hidden layers in a 14-28-28-48-25 unit constellation, see D.2. The hidden units have Sigmoid as activation function and the output unit has linear as activation function. The learning rate for the Adam algorithm is configured to 0.01. The batch size is quite large at 512. Dropout is completely disabled. The performance of this architecture ended up with a MAE of < 3% on the validation set and around 3% on the test set. Before the model can be interpreted and explained in more detail, the training curve of the best architecture is presented to confirm these observations with more data.

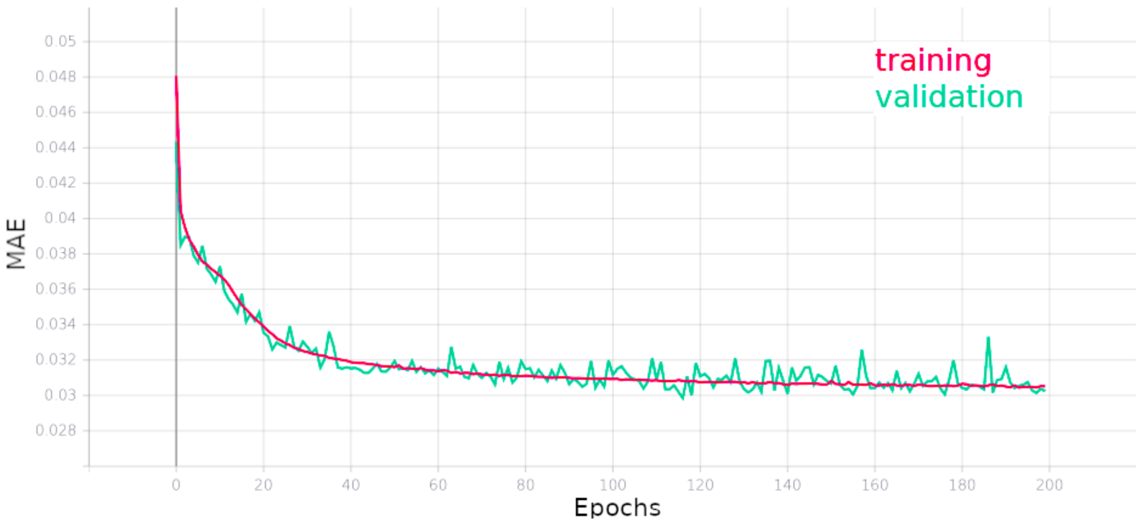

Figure 4.3: Training curve of the best architecture. The red line shows the MAE curve on training data and green on validation data.

The red curve shows the MAE on training data and the green curve the MAE on validation data. As expected, the performance of the model improves with increasing number of epochs. The curve of the validation data always oscillates around the training curve, which is a typical behaviour for the Adam algorithm. The plot shows that there isn't any distance between validation and training error, and thus it shows clearly that the model is not *overfitting* as described in 3.1.4, so indeed no dropout is required. This can be explained simply by the fact that sufficient data are available. Further, the training error curve flattened significantly up to epoch 200. However, as expected, the 200 epochs are not enough and the model is still *underfitted*. The error decrease is almost linear between epoch 40 and 200, and it can be assumed that the error will decrease for far more epochs. More information can be obtained from figure 4.4.

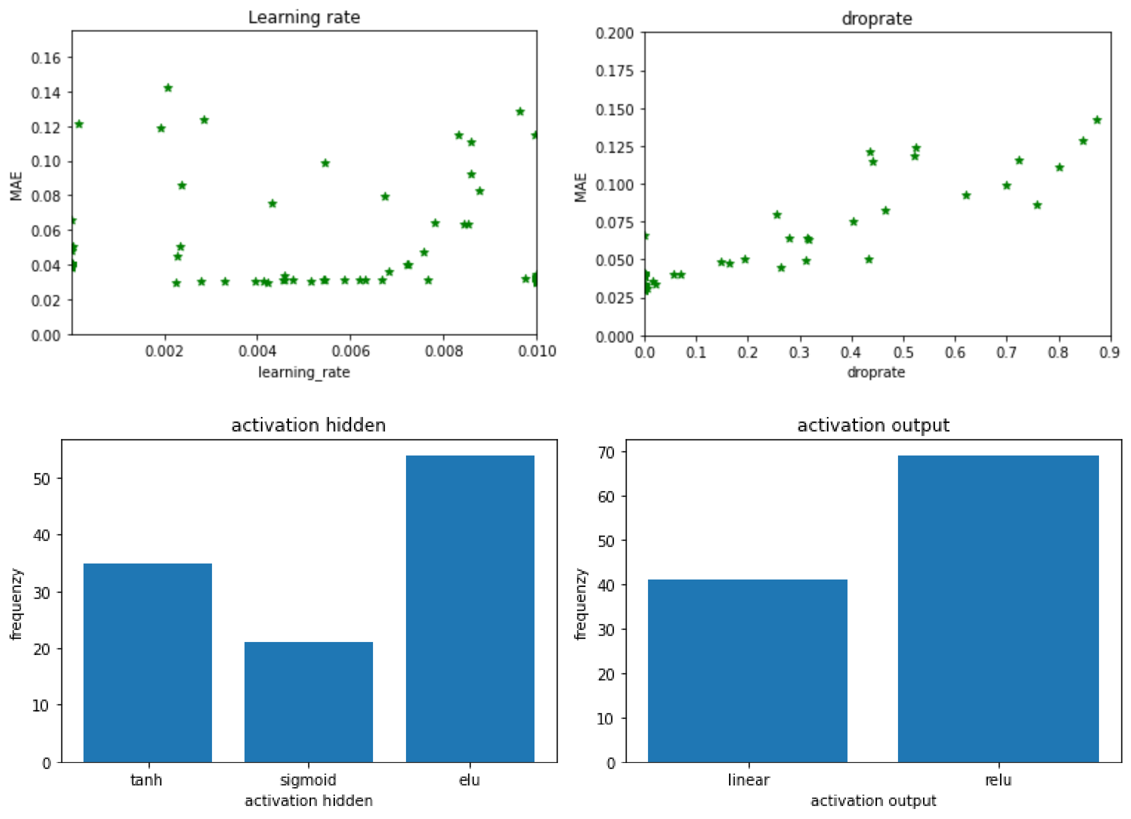

Figure 4.4: Additional plots from the model. Figure (a) shows the plot of all learning rates evaluations from the BO algorithm. The learning rate is plotted against MAE. Figure (b) shows the plot of all dropout rate evaluations from the BO algorithm. The dropout rate is plotted against MAE. The few outliers are removed from figure (a) and (b) to make the plot more clean. Figure (c) and (d) showing the histogram of used activation functions during the optimization process.

Source: own

The plots in figure 4.4 show more information about the model gathered from the BO algorithm. Both, figure 4.4 and the table in D.2 confirm the observations made above. The use of dropout is not suitable, indicated by the linear increase in 4.4 (b). However, this linear increase can flatten out as the number of epochs increases. The learning rate does not impact the performance significantly, as the distribution in 4.4 (a) is uniform.

Figure 4.4 (c) and (d) showing the accumulated number of used activation functions. Interestingly, these show a contradiction with the table in D.2. One would intuitively assume from the figure 4.4 (c) that Sigmoid does not perform very well compared to tanh and ELU as hidden activation function, since Sigmoid was less checked from the BO algorithm. However, the table D.2 show, that Sigmoid and ELU performed better than tanh. Similarly, with the output activation function. Table D.2 shows that Linear and ReLU perform similarly well, although Figure 1.4 (d) shows that the ReLU activation function has been far more checked from the algorithm.

Above, the results of the model were presented. The model was designed for five ambient temperature ranges as described in 4.1. An MAE of 3% was achieved. It is also interesting to know how well the model can predict the individual temperature ranges. This is interesting if one is looking for weaknesses of the model and is looking for further improvements of it, or if the user does only need a specific temperature range. For this purpose, the same dataset is split in five smaller ones after the data transformation step in 2.3.2 - for each ambient temperature set. Apart from that, all the steps are left the same as in 2.3. This evaluation is split in two parts:

- Performance of the model on specific ambient temperatures without training the model for the specific ambient temperature set
- Performance of the model trained specific for one ambient temperature set

The results of the first experiment are shown in table 4.2.

Table 4.2: The table shows the performance of the trained model on specific ambient temperatures without training for the specific ambient temperature set

Temp.	**MAE**
25 °C	0.019891
10 °C	0.026655
0 °C	0.026448
-10 °C	0.037762
-20 °C	0.057582

The best evaluation is the set with 25 °C ambient temperature, which has a MAE of < 2%, descending to the worst set with -20 °C ambient temperature, which has a MAE of around 5.7%. To explain this behaviour, one must take a look at the size of each sub-dataset. The sub-datasets contain 1050128, 928604, 682210, 552294, 462363 samples in that order. Obviously, the sub-datasets are very unbalanced. The behaviour of the model can be explained by the fact that the model tries to minimize the Metric. The model gets the best performance by reducing the error on the largest sub-dataset. On the one hand, this reduces the total error, on the other hand it has the side effect that the maximum peak error increases, which might be unsuitable for some applications. Balancing the sub-datasets could solve this problem. It should also have a positive effect if the model is designed larger, combined with more training effort. It is noticeable that the performance on the 0 °C data set is better than that on the 10 °C. Presumably, the sub-datasets vary in predictability, but from the current state of knowledge, this cannot be fully explained and needed more experiments.

The results of the second experiment are shown in table 4.3.

Table 4.3: The table shows the performance of the model specific trained for one ambient temperature set. However, the architecture is let the same as optimized by the BO algorithm.

Temp.	**MAE**
25 °C	0.0159539
10 °C	0.0237457
0 °C	0.0222675
-10 °C	0.0307379
-20 °C	0.0504489

Note that the architecture did not adjust for this experiment due to limited work time. Therefore, the model's performance improves, but not significantly. Furthermore, the behaviour is also the same as in experiment 1. The reason for that is that the architecture is still adapted to the entire dataset.

Chapter 5

Conclusion

The subject of this work is the design of a neural network specific for battery capacity estimation. However, many design steps in this work can also be transferred to other domains as well. All theoretical and practical topics are explained and discussed in depth. Further, the normalized State-of-Charge is introduced and used as capacity indicator for practical purposes, which scales the working range of the battery capacity in the range of 0% to 100%. Before the ANN is designed, a suitable dataset is selected and prepared on the basis of predefined criteria. The preprocessing tasks are done with Matlab. A Bayesian optimization strategy is elaborated to optimize the architecture of the ANN human independent. These tasks are implemented in Python. The ANN model is optimized sequentially in 110 iterations. After the experiment, the optimized model is able to estimate the SoC with an MAE of 3% on a dataset containing ambient temperatures for -20 °C up to +25 °C. The evaluation of the experiment shows that much better model performances are possible, however the experiment was limited due to lack of computational resources. Additional experiments show that the model does not predict the different ambient temperatures equally, which requires further experiments. Nevertheless, it can be said that the objectives of the work were successfully achieved.

Appendix A

Plot Of NASA Original Dataset

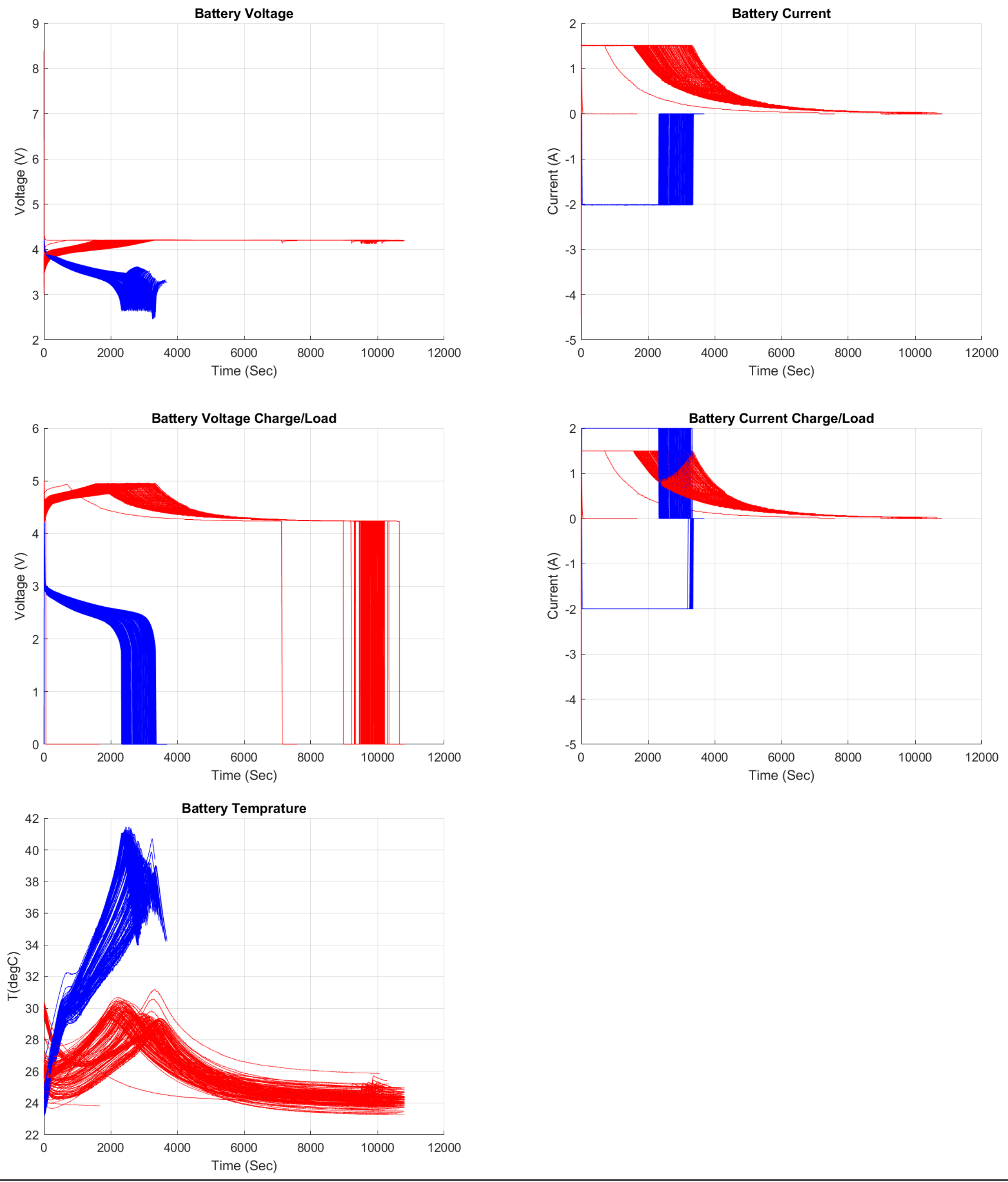

Figure A.1: NASA original *Battery Data Set* of the battery B0005 is plotted here, the download link [SG07]. The features are plotted over time in seconds. Features are terminal voltage, current, cell temperature, charge/load voltage and charge/load current. Blue is battery discharging and red is battery charging. It is the oldest battery dataset ever created. The dataset has many spikes in the data. The current and the ambient temperature are constant, which are in many cases not a real life scenario. The data set is more suitable for SOH prediction tasks.

Appendix B

Data Preprocessing: Matlab Code

B.1 Determine Normalized SoC

The codes can be found online at: `https://github.com/KocFatih/Thesis_BS`

```matlab
% This code concatenates the files and determines the normalized SOC
% The .mat file(s) has to be at root folder with this file
% To execute this file, the .mat file(s) has to be at root with this
    file. The regenerated file(s) will be put in <foldername> which has
     also be at root.
% Column order: voltage, current, temperature, ampere-hour, SOC, time

clc
clear
foldername = '1.determine_SoC';    % Save to folder
filename = 'Battery_Dataset_1';
files = dir('*.mat');

% Loading .mat files
for i = 1:numel(files)
    out(i) = load(files(i).name);
end

dataset = [];

% Concatenate the files
for k = 1:numel(out)
    tmp = length(dataset);
    dataset(k).data(:,1) = out(k).meas.Voltage;
    dataset(k).data(:,2) = out(k).meas.Current;
    dataset(k).data(:,3) = out(k).meas.Battery_Temp_degC;
    dataset(k).data(:,4) = out(k).meas.Ah;
    dataset(k).data(:,6) = out(k).meas.Time;
end

```

```
% Calculate: SOCn(t) = (Qk(t) - min(Qk)) / (max(Qk) - min(Qk))
for k = 1:numel(out)
    dataset(k).data(:,5) = (dataset(k).data(:,4)
                              - min(dataset(k).data(:,4)))
                              / (max(dataset(k).data(:,4))
                              - min(dataset(k).data(:,4)));
end

% Save to file
save(sprintf('%s\\%s\\%s\\',pwd,foldername,filename),'dataset');
```

B.2 Standardize Features

```
% This code concatenate all drive cycles and standardize the features
% Features and target are separated and only relevant features are
   saved
% To execute this file, the .mat file(s) has to be at root with this
   file. The regenerated file(s) will be put in <foldername> which has
    also be at root.

clc
clear
foldername = '2.normalize';    % Save to folder
filename = 'Battery_Dataset_2';
file = dir('*.mat');
load(file.name);

X = [];
Y = [];

% Concatenate all drive cycles
for m = 1:numel(dataset)
    % Order: voltage, current, temperature, ampere-hour, SoC, time,
    % X: voltage, current, temperature
    % Y: SoCn
    % Time: time      For timeseries tasks if needed!
    tmp = length(X);
    X(tmp+1:tmp+length(dataset(m).data(:,1)),1:3) =
       dataset(m).data(:,1:3);
    Y(tmp+1:tmp+length(dataset(m).data(:,1)),1) =
      dataset(m).data(:,5);
    % Time(tmp+1:tmp+length(dataset(m).data(:,1)),1) =
      dataset(m).data(:,6);
end

% z-score standardization
U = mean(X,1);
S = std(X,1);
Z = (X-U)./S;
X = Z;

%{
% Alternatively one can min-max normalize the data
% This block of code is for normalization.

% lambda
% lmd1 = 0.1;       %lower boundary e.g. 0.1
% lmd2 = 0.9;       %upper boundary e.g. 0.9

% min-max normalizing x_=lmd1+(x-min(x))/(max(x)-min(x))*(lmd2-lmd1)
% max_v    = max(X(:,1));
% min_v    = min(X(:,1));
% max_i    = max(X(:,2));
% min_i    = min(X(:,2));
% max_t    = max(X(:,3));
% min_t    = min(X(:,3));

% Voltage
% X(:,1) = lmd1+(X(:,1)-min_v)/(max_v-min_v)*(lmd2-lmd1);
% Current
% X(:,2) = lmd1+(X(:,2)-min_i)/(max_i-min_i)*(lmd2-lmd1);
% Temperature
% X(:,3) = lmd1+(X(:,3)-min_t)/(max_t-min_t)*(lmd2-lmd1);
%}

% Save to file
save(sprintf('%s\\%s\\%s\\',pwd,foldername,filename),'X', 'Y');
```

B.3 Data Partitioning

```matlab
% This code split the data in training/validation/test subset
% The subset sizes can be configured with test_size and val_size
% To execute this file, the .mat file(s) has to be at root with this
    file. The regenerated file(s) will be put in <foldername> which has
     also be at root.
clc
clear

% Configure subset sizes
test_size = 0.25;
val_size  = 0.15;

foldername = '3.partition';    % Save to folder
filename = 'Battery_Dataset';
file = dir('*.mat');
out = load(file.name);

% Random permutation
[m,n] = size(out.X);
rp = randperm(m);

% Partition test
X_test = out.X(rp(1:round(m*test_size)), :);
Y_test = out.Y(rp(1:round(m*test_size)), :);
% Partition validation
if (val_size ~= 0)
    X_val = out.X(rp(round(m*test_size)+1:
                     round(m*(test_size+val_size))), :);
    Y_val = out.Y(rp(round(m*test_size)+1:
                     round(m*(test_size+val_size))), :);
end
% Partition training
X_train = out.X(rp(round(m*(test_size+val_size))+1:end), :);
Y_train = out.Y(rp(round(m*(test_size+val_size))+1:end), :);

% Save to file
save(sprintf('%s\\%s\\%s\\',pwd,foldername,filename),
     'X_train','Y_train','X_test','Y_test','X_val','Y_val');
```

B.4 Plot Histogram

```matlab
% This code was used to create the histograms and don't modify the
    dataset. Depending on what is to be plotted, some parts of the code
     may need to be commented out

close all
%-----------------------------------------------
%                 histogram of data
%-----------------------------------------------
hold on;

% Plot histogram of output(SoC)
data = Y;
edges = linspace(0, 1, 20);

% Input without data normalization/standardization
data = X;
edges = linspace(-25, 35, 30);

% Plot with z-score
data = X;
edges = linspace(-4, 4, 40);

% Plot the histogram.
histogram(data(:,1), 'BinEdges',edges, 'FaceColor', 'r' );
histogram(data(:,2), 'BinEdges',edges, 'FaceColor', 'b');
histogram(data(:,3), 'BinEdges',edges, 'FaceColor', 'g');

legend('voltage','current','temp')
xlabel('space')
ylabel('Frequency')
hold off;

%-----------------------------------------------
%            histogram of partitions
%-----------------------------------------------

edges = linspace(0, 1, 20);
hold on;
```

```
histogram(Y_train, 'BinEdges',edges, 'FaceColor', 'r' );
histogram(Y_test, 'BinEdges',edges, 'FaceColor', 'b');
histogram(Y_val, 'BinEdges',edges, 'FaceColor', 'g');
legend('train','test','val')
xlabel('SoC')
ylabel('Frequency')
hold off;

%-----------------------------------------------
%                   save plot
%-----------------------------------------------

% Save plot
set(gcf);
saveas(gcf, 'histogram_partitions.png');
```

B.5 Plot Dataset

```
% This code was used to plot the data and don't modify the dataset.
    Depending on what is to be plotted, some parts/sections of the code
     may need to be commented out. The dataset must be loaded into
    Matlab in order to plot it.

close all
%-----------------------------------------------
%          Plot Original Dataset (only)
%-----------------------------------------------
% This code section allows to plot the unmodified original dataset.

figure
hold on
plot(meas.Time,meas.Current,'r')
hold off
grid on
title ('Battery Current')
xlabel('Time (Sec)')
ylabel('Current(A)')


```

```matlab
figure
hold on
plot(meas.Time,meas.Voltage,'r')
hold off
grid on
title ('Battery Voltage')
xlabel('Time (Sec)')
ylabel('Voltage (V)')

figure
hold on
plot(meas.Time,meas.Power,'r')
hold off
grid on
title ('Battery Power')
xlabel('Time (Sec)')
ylabel('Power(W)')

figure
hold on
plot(meas.Time,meas.Ah,'r')
hold off
grid on
title ('Battery Amp-Hours')
xlabel('Time (Sec)')
ylabel('Amp-Hours(Ah)')

figure
hold on
plot(meas.Time,meas.Battery_Temp_degC,'r')
hold off
grid on
title ('Battery Temperature')
xlabel('Time (Sec)')
ylabel('T(degC)')
```

```
%--------------------------------------------------
%               Plot Modified Data
%--------------------------------------------------
% This code is only usable, after the execution of the code at 'B.1
    Determine Normalized SoC'! The dataset contains 45 separate drive
    cycles (25, 10, 0, -10, -20 in that order). Enter the wished drive
    cycles in the array to plot them:

print=[5 20 30 44];

% E.g. this plotting the cycles ...
%5 = 25degC_US06
%20= 0degC_Cycle_2
%30= n10degC_UDDS
%44= n20degC_Cycle_4

% Voltage
figure
grid on
title ('Battery Voltage')
xlabel('Time (Sec)')
ylabel('Voltage (V)')
hold on
for i=print
    plot(dataset(i).data(:,6),dataset(i).data(:,1))
end
legend('25degC US06','0degC Cycle2','n10degC UDDS', 'n20degC Cycle4')
hold off

set(gcf);
%saveas(gcf, 'battery_voltage.png');

%current
figure
grid on
title ('Battery Current')
xlabel('Time (Sec)')
ylabel('Current (A)')
hold on

```

```
for i=print
    plot(dataset(i).data(:,6),dataset(i).data(:,2))
end
legend('25degC US06','0degC Cycle2','n10degC UDDS', 'n20degC Cycle4')
hold off

%saveas(gcf, 'battery_current.png');

%temperature
figure
grid on
title ('Battery Temperature')
xlabel('Time (Sec)')
ylabel('T (degC)')
hold on
for i=print
    plot(dataset(i).data(:,6),dataset(i).data(:,3))
end
legend('25degC US06','0degC Cycle2','n10degC UDDS', 'n20degC Cycle4')
hold off

%saveas(gcf, 'battery_Temp.png');

%Amp-Hours (electric charge)
figure
grid on
title ('Battery Amp-Hours')
xlabel('Time (Sec)')
ylabel('Amp-Hours(Ah)')
hold on
for i=print
    plot(dataset(i).data(:,6),dataset(i).data(:,4))
end
legend('25degC US06','0degC Cycle2','n10degC UDDS', 'n20degC Cycle4')
hold off

%saveas(gcf, 'battery_Amp-Hours.png');



```

```
%SoC
figure
grid on
title ('Battery SoC')
xlabel('Time (Sec)')
ylabel('SoC_N (%)')
hold on
for i=print
    plot(dataset(i).data(:,6),dataset(i).data(:,5))
end
legend('25degC US06','0degC Cycle2','n10degC UDDS', 'n20degC Cycle4')
hold off

%saveas(gcf, 'battery_SoC.png');
```

Appendix C

Design Model: Python Code

This code can be found online at: `https://github.com/KocFatih/Thesis_BS`

```python
# This code implementing an forward neural network model optimized with an powerful global bayesian optimization strategy.

import numpy as np
from scipy import io
from matplotlib import pyplot as plt
import pickle
import keras
from keras.models import Sequential
from keras.layers import Dense, Input, Dropout
from keras import backend
from tensorflow.keras.optimizers import Adam
from tensorflow.python.keras.callbacks import TensorBoard
from keras.models import load_model
from keras.callbacks import EarlyStopping

import GPy
import GPyOpt
from GPyOpt.methods import BayesianOptimization
from GPy.kern import Matern52


num_nodes = 50  # Max possible nodes/units
epochs = 200    # Epochs for neural network
max_iter = 95   # Bayesian optimization iterations

# To display the iterations
iteration = 0  # Don't change
# Best found performance of the model so far
best_mae = 100 # Don't change


```

```
# Bounds dict for search space
bounds = [
           {'name': 'num_hidden_nodes_1', 'type': 'discrete',
            'domain': tuple(np.array(range(3,num_nodes+1))) },
           {'name': 'num_hidden_nodes_2', 'type': 'discrete',
            'domain': tuple(np.array(range(num_nodes+1))) },
           {'name': 'num_hidden_nodes_3', 'type': 'discrete',
            'domain': tuple(np.array(range(num_nodes+1))) },
           {'name': 'num_hidden_nodes_4', 'type': 'discrete',
            'domain': tuple(np.array(range(num_nodes+1))) },
           {'name': 'num_hidden_nodes_5', 'type': 'discrete',
            'domain': tuple(np.array(range(num_nodes+1))) },
           {'name': 'learning_rate', 'type': 'continuous',
            'domain': (1e-6, 1e-2) },
           {'name': 'activation_hidden', 'type': 'categorical',
            'domain': (0, 1, 2) }, #tanh, sigmoid, elu
           {'name': 'activation_output', 'type': 'categorical',
            'domain': (0, 1) }, #linear, relu
           {'name': 'dropout', 'type': 'continuous',
            'domain': (0.0, 0.9) },
           {'name': 'batch_size', 'type': 'discrete',
            'domain': (64, 128, 256, 512, 1024) }
         ] # End bounds

# Constraints for hidden layer
constraints = [
                {'name': 'constr_layer_2',
                 'constraint': 'x[:,1] - x[:,1] * x[:,0]'},
                {'name': 'constr_layer_3',
                 'constraint': 'x[:,2] - x[:,2] * x[:,1]'},
                {'name': 'constr_layer_4',
                 'constraint': 'x[:,3] - x[:,3] * x[:,2]'},
                {'name': 'constr_layer_5',
                 'constraint': 'x[:,4] - x[:,4] * x[:,3]'}
              ] # End constraints
# One can insert fixed points to the Gaussian process
#init_param =   [
#                 [10, 20, 30, 40, 50, 5e-4, 1, 0, 0.2, 512],
#                 [40, 0, 0, 0, 0, 1e-4, 1, 1, 0.1, 128]
#               ]
```

```python

# Load data from location
data = io.loadmat('~/Documents/datasets/Battery_Dataset.mat')

X_test = data['X_test']
X_train = data['X_train']
X_val = data['X_val']
Y_test = data['Y_test']
Y_train = data['Y_train']
Y_val = data['Y_val']


#SoC class, containing the ANN model and some helper functions
class SOC():
    def __init__(self, num_hidden_nodes_1=1, num_hidden_nodes_2=0,
                 num_hidden_nodes_3=0, num_hidden_nodes_4=0,
                 num_hidden_nodes_5=0, learning_rate=0,
                 activation_hidden='sigmoid',
                 activation_output='linear',
                 dropout=0.0):

        self.num_hidden_nodes_1=num_hidden_nodes_1
        self.num_hidden_nodes_2=num_hidden_nodes_2
        self.num_hidden_nodes_3=num_hidden_nodes_3
        self.num_hidden_nodes_4=num_hidden_nodes_4
        self.num_hidden_nodes_5=num_hidden_nodes_5
        self.learning_rate=learning_rate
        self.activation_hidden=activation_hidden
        self.activation_output=activation_output
        self.dropout=dropout

        self.model = self.create_model()
        self.history = None
        self.score = None
        self.path_best_model = 'best_model.h5'

    #Create the dynamic Model
    def create_model(self):
        model = Sequential()

        # Input-layer
        model.add(Input(shape=(X_train.shape[1],)))

        # Build layer 1
        model.add(Dense(self.num_hidden_nodes_1,
                  activation=self.activation_hidden, use_bias='true'))

        # Build layer 2
        if self.num_hidden_nodes_2 != 0:
            model.add(Dropout(self.dropout))
            model.add(Dense(self.num_hidden_nodes_2,
                  activation=self.activation_hidden, use_bias='true'))

        # Build layer 3
        if self.num_hidden_nodes_3 != 0:
            model.add(Dropout(self.dropout))
            model.add(Dense(self.num_hidden_nodes_3,
                  activation=self.activation_hidden, use_bias='true'))

        # Build layer 4
        if self.num_hidden_nodes_4 != 0:
            model.add(Dropout(self.dropout))
            model.add(Dense(self.num_hidden_nodes_4,
                  activation=self.activation_hidden, use_bias='true'))

        # Build layer 5
        if self.num_hidden_nodes_5 != 0:
            model.add(Dropout(self.dropout))
            model.add(Dense(self.num_hidden_nodes_5,
                  activation=self.activation_hidden, use_bias='true'))

        # Output layer. One output for a regression problem
        model.add(Dense(1, activation=self.activation_output,
                  use_bias='true'))

        # Adam optimizer
        optimizer = Adam(lr=self.learning_rate)



        # Compile Model
        model.compile(optimizer=optimizer,
                      loss='mean_squared_error',
                      metrics=[keras.metrics.MeanAbsoluteError(),
                               keras.metrics.MeanSquaredError(),
                               keras.metrics.RootMeanSquaredError()])

        return model
        # End create_model method

    # Start training process
    def fit(self, X_train, Y_train, batch_size, epochs, X_val, Y_val):

        # Configurate Early Stopping for 30 iterations
        es = EarlyStopping(monitor='val_loss', mode='min',
                           verbose=0, patience=30)

        # Foldername for Tensorboard logs
        log_dir = "logs/" + str([self.num_hidden_nodes_1,
                                 self.num_hidden_nodes_2,
                                 self.num_hidden_nodes_3,
                                 self.num_hidden_nodes_4,
                                 self.num_hidden_nodes_5,
                                 self.learning_rate,
                                 self.activation_hidden,
                                 self.activation_output,
                                 self.dropout,
                                 batch_size
                                 ])

        # Tensorboard callback. Runs every epoch.
        # To see the Tensorboard logs ...
        # ... run "tensorboard --logdir=logs/" in the terminal ...
        # ... and open the url: http://localhost:6006/
        callback_log = TensorBoard(log_dir=log_dir)

        # Start training
        self.history = self.model.fit(
                                    X_train,
                                    Y_train,
                                    batch_size=batch_size,
                                    epochs=epochs,
                                    callbacks=[es,
                                               callback_log],
                                    validation_data=(X_val, Y_val))

        # End fit method


    # Return the metrics
    def get_metric(self):
        #return self.history.history['val_mean_squared_error'][-1]
        return [min(self.history.history['val_mean_absolute_error']),
                min(self.history.history['val_mean_squared_error']),
                min(self.history.history['val_root_mean_squared_error'
            ])]

    # Save model
    def save(self):
        self.model.save(self.path_best_model)

    # Loading the model specified in path_best_model
    def load_best(self):
        self.model = load_model(self.path_best_model)

    # Evaluation with test set. Get the value with get_score() method
    def evaluate(self, X_test, Y_test):
        self.score = self.model.evaluate(X_test, Y_test, verbose=0)

    # Get score after method evaluation(.) is finished
    def get_score(self):
        return self.score[1]

    # End SoC class

# Saves the iterated hyperparameters and there metrics in a file
def save_model(hyperparam, metric):
    global iteration
    file = open("save_model.txt", "a")

    file.write(
        "{0}:\n{1} {2} {3} {4} {5} {6} {7} {8} {9} {10}\n".format(
                                iteration,
                                int(hyperparam[:,0]),
                                int(hyperparam[:,1]),
                                int(hyperparam[:,2]),
                                int(hyperparam[:,3]),
                                int(hyperparam[:,4]),
                                float(hyperparam[:,5]),
                                int(hyperparam[:,6]),
                                int(hyperparam[:,7]),
                                float(hyperparam[:,8]),
                                int(hyperparam[:,9])
                                                                 )
                ) #end write

    file.write("{0} {1} {2}".format(metric[0], metric[1], metric[2]))
    file.write("\n")
    file.close()
    # End save_model function

# Noisy objective function
def objective_func(hyperparam):
    print('--------------------------------------------------')
    print('------------- Start iteration ---------------------')
    global iteration
    global best_mae
    print("iteration: {0}".format(iteration+1))
    print(hyperparam)

    # Clear Tensorflow session (Tensorflow-graph), freeing memory.
    backend.clear_session()

    # Convert from numpy.ndarray to elementary data types
    num_hidden_nodes_1 = int(hyperparam[0,0])
    num_hidden_nodes_2 = int(hyperparam[0,1])
    num_hidden_nodes_3 = int(hyperparam[0,2])
    num_hidden_nodes_4 = int(hyperparam[0,3])
    num_hidden_nodes_5 = int(hyperparam[0,4])
    learning_rate      = float(hyperparam[0,5])
    activation_hidden   = int(hyperparam[0,6])
    activation_output   = int(hyperparam[0,7])
    dropout             = float(hyperparam[0,8])
    batch_size          = int(hyperparam[0,9])

    # Wrapper for activation functions
    if activation_hidden == 0:
        activation_hidden = 'tanh'
    elif activation_hidden == 1:
        activation_hidden = 'sigmoid'
    else:
        activation_hidden = 'elu'

    if activation_output == 0:
        activation_output = 'linear'
    else:
        activation_output = 'relu'

    # Initialize new model
    soc = SOC(num_hidden_nodes_1=num_hidden_nodes_1,
              num_hidden_nodes_2=num_hidden_nodes_2,
              num_hidden_nodes_3=num_hidden_nodes_3,
              num_hidden_nodes_4=num_hidden_nodes_4,
              num_hidden_nodes_5=num_hidden_nodes_5,
              learning_rate=learning_rate,
              activation_hidden=activation_hidden,
              activation_output=activation_output,
              dropout=dropout)

    # Train model
    soc.fit(X_train = X_train,
            Y_train = Y_train,
            X_val = X_val,
            Y_val = Y_val,
            batch_size = batch_size,
            epochs = epochs)

    # Get validation metrics
    metric = soc.get_metric()

    # Check if the new evaluation was better then the best so far at
   some point and if so it becomes the new best and is saved
    if metric[0] < best_mae:  #metric[0] is mae
        soc.save()
        best_mae = metric[0]

    # Deleting the model from keras model
    del soc.model

    iteration = iteration+1
    # Save evaluation data
    save_model(hyperparam, metric)

    print('+++++++++++++ End iteration +++++++++++++++++++++++')
    return metric[0]
    # End objective function

# Reset log file
file = open("save_model.txt", "w")
file.write("Iteration: hid_nod_1 hid_nod_2 hid_nod_3 hid_nod_4
   hid_nod_5 lea_rate act_hid act_out dropout batch_size\n")
file.close()

# Matern 5/2 - Kernel is already default
# kernel = Matern52(input_dim=len(bounds))

# Configurate Bayesian optimization using Gaussian Processes.
opt = GPyOpt.methods.BayesianOptimization(
                    f=objective_func,
                    domain=bounds,
                    constraints=constraints,
                    #kernel=kernel,
                    #X=np.array(init_param), # Initial set
                    #Y=Y,
                    initial_design_numdata=15, # Initial Random points
                    maximize=False  # Find minimum
                )



```

```
# Run optimization
opt.run_optimization(max_iter=max_iter,
                     evaluations_file='eval.txt',
                     report_file='report.txt',
                     models_file='models.txt')


print('-------------------------------------------------')
print('-------------------------------------------------')
print('--------------Optimization finished--------------')
print('-------------------------------------------------')
print('-------------------------------------------------')

# Plot of optimization. It shows the progress
opt.plot_convergence()

# Display all tested values
print(opt.X[:])
print(opt.Y[:])
print(opt.x_opt) #best set
print(opt.fx_opt) #best result


# Sorted list of evaluations
opt_list = []

for i in range(len(opt.Y)):
    opt_list.append([opt.Y[i][0], opt.X[i].tolist()])

opt_list = sorted(opt_list, key=lambda x: x[0])

file = open("sorted_save_model.txt", "w")
file.write("[loss, [hid_nod_1 hid_nod_2 hid_nod_3 hid_nod_4 hid_nod_5
    lea_rate act_hid act_out dropout batch_size]]\n")
for element in opt_list:
    file.write(str(element) + "\n")

file.close()
# End sorted list

# Save/load the evaluation data
x = []
y = []

# Save
with open("myData_x", "wb") as fp:
    pickle.dump(opt.X[:], fp)
with open("myData_y", "wb") as fp:
    pickle.dump(opt.Y[:], fp)

# Load
# with open("myData_x", "rb") as fp:
#     x = pickle.load(fp)
# with open("myData_y", "rb") as fp:
#     y = pickle.load(fp)

# Load the best model and evaluate on test set
soc = SOC()
soc.load_best()
score = soc.evaluate(X_test, Y_test)
print('Evaluation on test set:', soc.get_score())
```

Appendix D

Data Of All Evaluations

D.1 Unsorted Data

The table shows all evaluations in original order. All hyperparameters are recorded containing the nodes of each five hidden layer (HN), the learning rate, activation of the hidden layers (AH), activation of the output layer (AO), dropout rate and batch size. The recorded metrics are MAE, MSE and RMSE. The numbers 0, 1, 2 in column AH denoting the tanh, Sigmoid and ELU activation function. The number 0 in column AO denoting linear and 1 denoting ReLU activation function.

All evaluations

Iter.	MAE	MSE	RMSE	HN1	HN2	HN3	HN4	HN5	learn. rate	AH	AO	dropout	batch size
1	0.111001	0.020945	0.144725	45	19	6	39	12	0.008571	1	1	0.801601	256
2	0.128675	0.023489	0.153264	45	19	6	20	20	0.009632	1	1	0.846676	512
3	0.123944	0.023429	0.153066	47	49	26	3	26	0.002863	0	1	0.523044	256
4	0.035938	0.002841	0.053309	14	34	49	47	17	0.006826	0	1	0.015673	256
5	0.142495	0.029819	0.172681	16	14	22	12	30	0.002063	0	1	0.874191	1024

6	0.079563	0.009243	0.096141	30	47	3	35	29	0.006733	1	1	0.256143	128
7	0.115073	0.020455	0.143021	50	1	49	31	3	0.008314	2	1	0.442067	64
8	0.049623	0.004402	0.066348	31	44	12	10	40	2.35e-05	2	0	0.310702	128
9	0.086085	0.013080	0.114368	23	45	36	31	41	0.002386	0	1	0.757406	64
10	0.099071	0.015843	0.125870	25	16	5	32	2	0.005456	2	0	0.698341	128
11	0.075260	0.011407	0.106804	28	31	16	40	12	0.004324	0	1	0.403866	512
12	0.121246	0.021487	0.146586	17	41	8	2	12	0.000177	1	0	0.435642	64
13	0.034006	0.002550	0.050506	21	30	35	47	23	0.004585	2	1	0.020596	512
14	0.063845	0.007468	0.086418	8	35	8	34	25	0.008539	1	1	0.314492	512
15	0.118661	0.021722	0.147385	34	11	10	12	24	0.001924	0	1	0.522646	512
16	0.092507	0.015272	0.123582	7	43	9	31	29	0.008595	1	0	0.619949	512
17	0.115347	0.020338	0.142613	12	38	8	31	29	0.009954	1	0	0.723308	512
18	0.064028	0.007071	0.084094	8	37	49	45	14	0.007805	1	0	0.280808	256
19	0.029883	0.002237	0.047307	20	30	35	48	22	0.004220	2	1	0.0	512
20	0.030581	0.002279	0.047747	19	30	31	47	20	0.003953	2	1	0.0	512
21	0.031135	0.002328	0.048251	14	30	34	48	22	0.004591	2	1	0.0	512
22	0.493399	0.331399	0.575672	17	34	35	50	17	0.001101	1	1	0.0	512
23	0.031045	0.002302	0.047986	19	29	33	47	22	0.004754	2	1	0.005531	512
24	0.031427	0.002364	0.048628	16	29	34	47	24	0.005442	2	1	0.0	512
25	0.031106	0.002328	0.048255	15	29	31	47	22	0.005414	2	1	0.0	512
26	0.031286	0.002354	0.048523	13	28	33	48	23	0.006206	2	1	0.0	512

27	0.030661	0.002284	0.047801	13	29	34	45	23	0.004141	2	1	0.0	512
28	0.030881	0.002341	0.048385	20	29	29	46	21	0.005159	2	1	0.0	512
29	0.030992	0.002333	0.048302	13	31	33	47	25	0.006329	2	1	0.0	512
30	0.030939	0.002310	0.048069	21	28	35	50	23	0.005858	2	1	0.0	512
31	0.039894	0.003338	0.057782	21	29	31	49	23	1e-06	2	0	0.0	512
32	0.391099	0.224826	0.474158	13	29	37	48	25	1e-06	2	0	0.9	512
33	0.047440	0.004319	0.065720	11	35	49	46	17	0.007570	0	0	0.165438	256
34	0.034189	0.002598	0.050970	14	30	32	46	23	0.01	2	1	0.0	512
35	0.493399	0.331399	0.575672	13	35	49	47	14	0.006523	1	1	0.005603	256
36	0.040368	0.003344	0.057830	14	34	49	47	19	0.007239	0	1	0.070291	256
37	0.050411	0.004688	0.068469	32	43	12	10	38	0.002341	2	0	0.432325	128
38	0.029974	0.002200	0.046905	15	29	32	48	23	0.002260	2	1	0.0	512
39	0.032282	0.002487	0.049876	22	29	33	48	22	0.01	2	1	0.0	512
40	0.031712	0.002380	0.048787	20	30	33	49	24	0.01	2	1	0.0	512
41	0.032214	0.002472	0.049724	15	28	33	46	21	0.009761	2	1	0.0	512
42	0.032168	0.002441	0.049408	18	27	30	48	21	0.01	2	1	0.0	512
43	0.045045	0.003716	0.060963	33	46	15	11	39	0.002275	2	0	0.264817	128
44	0.039846	0.003367	0.058033	12	29	32	47	21	1e-06	2	1	0.0	512
45	0.040023	0.003391	0.058235	17	28	31	46	24	1e-06	0	1	0.0	512
46	0.040092	0.003248	0.056995	13	33	49	46	18	0.007224	0	0	0.057640	256
47	0.063438	0.007558	0.086940	9	35	49	45	19	0.008430	0	0	0.318482	256

48	0.030379	0.002248	0.047415	17	31	32	46	25	0.002802	2	1	0.0	512
49	0.031529	0.002398	0.048972	13	32	30	47	26	0.007651	2	1	0.0	512
50	0.050497	0.004503	0.067110	35	43	14	9	41	2.98e-05	2	0	0.193868	128
51	0.032001	0.002421	0.049212	18	30	29	48	23	0.01	2	1	0.0	512
52	0.040691	0.003486	0.059042	21	27	31	47	19	1e-06	0	1	0.0	512
53	0.039985	0.003374	0.058091	17	29	28	46	19	1e-06	0	1	0.0	512
54	0.039634	0.003341	0.057804	12	32	32	44	26	1e-06	2	1	0.0	512
55	0.030563	0.002263	0.047571	11	28	30	46	25	0.003310	2	1	0.0	512
56	0.033246	0.002534	0.050342	14	29	30	45	27	0.01	2	1	0.0	512
57	0.031227	0.002350	0.048486	13	26	32	43	24	0.006669	2	1	0.0	512
58	0.048401	0.004283	0.065449	31	43	17	8	39	1e-06	0	0	0.147656	128
59	0.038946	0.003188	0.056466	32	42	16	13	40	2.02e-06	0	0	0.0	128
60	0.032673	0.002466	0.049663	13	25	30	46	23	0.01	2	1	0.0	512
61	0.039584	0.003321	0.057633	16	29	33	42	24	1e-06	2	1	0.0	512
62	0.039807	0.003348	0.057867	15	27	27	43	23	1e-06	2	1	0.0	512
63	0.040557	0.003457	0.058798	11	29	31	42	23	1e-06	0	1	0.0	512
64	0.032187	0.002460	0.049603	14	28	26	48	23	0.01	2	1	0.0	512
65	0.040795	0.003499	0.059153	13	26	26	46	26	1e-06	0	1	0.0	512
66	0.030113	0.002238	0.047311	17	26	26	47	22	0.01	1	0	0.0	512
67	0.082729	0.012503	0.111820	5	38	49	44	15	0.008767	0	0	0.465065	256
68	0.485474	0.325112	0.570186	19	28	25	49	21	1e-06	2	1	0.9	512

69	0.039224	0.003250	0.057010	34	42	17	11	37	1e-06	2	1	0.0	128
70	0.030731	0.002244	0.047379	16	26	27	46	22	0.01	1	0	0.0	512
71	0.032693	0.002491	0.049912	18	28	30	45	20	0.01	2	1	0.0	512
72	0.030417	0.002238	0.047310	16	24	26	47	24	0.01	1	0	0.0	512
73	0.029855	0.002202	0.046926	14	28	28	48	25	0.01	1	0	0.0	512
74	0.032795	0.002524	0.050246	12	29	27	46	23	0.01	0	1	0.0	512
75	0.493399	0.331399	0.575672	11	26	27	49	23	0.01	1	1	0.0	512
76	0.031562	0.002423	0.049228	15	29	27	46	24	0.01	2	1	0.0	512
77	0.493399	0.331399	0.575672	33	43	18	11	41	0.01	2	1	0.9	128
78	0.040261	0.003417	0.058459	32	42	15	13	37	1e-06	0	0	0.0	128
79	0.033128	0.002541	0.050408	15	27	27	47	25	0.01	0	0	0.0	512
80	0.040090	0.003341	0.057805	34	43	15	9	36	1e-06	0	0	0.0	128
81	0.040737	0.003482	0.059011	13	28	30	45	24	1e-06	0	1	0.0	512
82	0.039717	0.003298	0.057433	32	41	14	14	40	1e-06	0	0	0.0	128
83	0.031489	0.002367	0.048653	13	30	27	47	24	0.01	2	0	0.0	512
84	0.038946	0.003231	0.056842	35	41	16	11	35	1e-06	2	1	0.0	128
85	0.030404	0.002256	0.047502	19	29	30	46	22	0.01	1	0	0.0	512
86	0.033080	0.002546	0.050466	16	24	30	46	24	0.01	2	1	0.0	512
87	0.040166	0.003363	0.057995	30	44	15	8	37	1e-06	0	0	0.0	128
88	0.031135	0.002350	0.048483	12	29	27	44	26	0.004553	0	1	0.0	512
89	0.039595	0.003292	0.057383	35	46	12	10	39	1e-06	0	0	0.0	128

90	0.032585	0.002508	0.050085	14	25	28	44	26	0.01	2	1	0.0	512
91	0.032909	0.002522	0.050224	13	29	24	45	24	0.01	0	1	0.0	512
92	0.030108	0.002236	0.047296	16	29	29	48	26	0.01	1	0	0.0	512
93	0.039890	0.003329	0.057705	31	40	17	9	36	1e-06	0	0	0.0	128
94	0.040583	0.003436	0.058619	33	43	13	6	39	1e-06	0	0	0.0	128
95	0.031705	0.002420	0.049196	14	24	32	45	21	0.01	2	1	0.0	512
96	0.030176	0.002208	0.046999	16	25	24	45	25	0.01	1	0	0.0	512
97	0.065928	0.008758	0.093587	14	28	26	46	28	1e-06	1	0	0.0	512
98	0.033181	0.002517	0.050178	13	30	27	44	21	0.01	0	1	0.0	512
99	0.039343	0.003245	0.056969	33	43	18	11	34	1e-06	2	0	0.0	128
100	0.030474	0.002252	0.047463	18	23	26	45	23	0.01	1	0	0.0	512
101	0.039517	0.003272	0.057207	30	42	17	5	38	1e-06	0	0	0.0	128
102	0.039451	0.003268	0.057171	37	42	11	8	40	1e-06	2	0	0.0	128
103	0.039525	0.003254	0.057045	33	47	14	12	36	1e-06	2	0	0.0	128
104	0.033413	0.002608	0.051077	10	28	30	43	27	0.01	2	1	0.0	512
105	0.032376	0.002517	0.050172	21	31	30	44	22	0.01	0	1	0.0	512
106	0.033253	0.002602	0.051018	10	32	28	44	24	0.01	0	1	0.0	512
107	0.039324	0.003240	0.056922	36	44	14	12	37	1e-06	2	0	0.0	128
108	0.040918	0.003541	0.059508	13	25	24	42	26	1e-06	0	1	0.0	512
109	0.030307	0.002247	0.047411	20	24	32	47	22	0.01	1	0	0.0	512

110	0.033124	0.002563	0.050628	10	31	31	45	24	0.01	2	1	0.0	512

Table D.1: Unsorted data of all evaluations

D.2 Sorted Data

The table shows all evaluations sorted by performance in descending order. All hyperparameters are recorded containing the nodes of each five hidden layer (HN), the learning rate, activation of the hidden layers (AH), activation of the output layer (AO), dropout rate and batch size and the metric MAE. The numbers 0, 1, 2 in column AH denoting the tanh, Sigmoid and ELU activation function. The number 0 in column AO denoting linear and 1 denoting ReLU activation function.

All evaluations sorted by performance in descending order										
MAE	HN1	HN2	HN3	HN4	HN5	LR	AH	AO	dropout	BS
0.029855	14	28	28	48	25	0.01	1	0	0.0	512
0.029883	20	30	35	48	22	0.004220	2	1	0.0	512
0.029974	15	29	32	48	23	0.002260	2	1	0.0	512
0.030108	16	29	29	48	26	0.01	1	0	0.0	512
0.030113	17	26	26	47	22	0.01	1	0	0.0	512
0.030176	16	25	24	45	25	0.01	1	0	0.0	512
0.030307	20	24	32	47	22	0.01	1	0	0.0	512
0.030379	17	31	32	46	25	0.002802	2	1	0.0	512

0.030404	19	29	30	46	22	0.01	1	0	0.0	512
0.030417	16	24	26	47	24	0.01	1	0	0.0	512
0.030474	18	23	26	45	23	0.01	1	0	0.0	512
0.030563	11	28	30	46	25	0.003310	2	1	0.0	512
0.030581	19	30	31	47	20	0.003953	2	1	0.0	512
0.030661	13	29	34	45	23	0.004141	2	1	0.0	512
0.030731	16	26	27	46	22	0.01	1	0	0.0	512
0.030881	20	29	29	46	21	0.005159	2	1	0.0	512
0.030939	21	28	35	50	23	0.005858	2	1	0.0	512
0.030992	13	31	33	47	25	0.006329	2	1	0.0	512
0.031045	19	29	33	47	22	0.004754	2	1	0.00553	512
0.031106	15	29	31	47	22	0.005414	2	1	0.0	512
0.031135	14	30	34	48	22	0.004591	2	1	0.0	512
0.031135	12	29	27	44	26	0.004553	0	1	0.0	512
0.031227	13	26	32	43	24	0.006669	2	1	0.0	512
0.031286	13	28	33	48	23	0.006206	2	1	0.0	512
0.031427	16	29	34	47	24	0.005442	2	1	0.0	512
0.031489	13	30	27	47	24	0.01	2	0	0.0	512
0.031529	13	32	30	47	26	0.007651	2	1	0.0	512
0.031562	15	29	27	46	24	0.01	2	1	0.0	512
0.031705	14	24	32	45	21	0.01	2	1	0.0	512

0.031712	20	30	33	49	24	0.01	2	1	0.0	512
0.032001	18	30	29	48	23	0.01	2	1	0.0	512
0.032168	18	27	30	48	21	0.01	2	1	0.0	512
0.032187	14	28	26	48	23	0.01	2	1	0.0	512
0.032214	15	28	33	46	21	0.009761	2	1	0.0	512
0.032282	22	29	33	48	22	0.01	2	1	0.0	512
0.032376	21	31	30	44	22	0.01	0	1	0.0	512
0.032585	14	25	28	44	26	0.01	2	1	0.0	512
0.032673	13	25	30	46	23	0.01	2	1	0.0	512
0.032693	18	28	30	45	20	0.01	2	1	0.0	512
0.032795	12	29	27	46	23	0.01	0	1	0.0	512
0.032909	13	29	24	45	24	0.01	0	1	0.0	512
0.033080	16	24	30	46	24	0.01	2	1	0.0	512
0.033124	10	31	31	45	24	0.01	2	1	0.0	512
0.033128	15	27	27	47	25	0.01	0	0	0.0	512
0.033181	13	30	27	44	21	0.01	0	1	0.0	512
0.033246	14	29	30	45	27	0.01	2	1	0.0	512
0.033253	10	32	28	44	24	0.01	0	1	0.0	512
0.033413	10	28	30	43	27	0.01	2	1	0.0	512
0.034006	21	30	35	47	23	0.004585	2	1	0.020596	512
0.034189	14	30	32	46	23	0.01	2	1	0.0	512

0.035938	14	34	49	47	17	0.006826	0	1	0.015673	256
0.038946	35	41	16	11	35	1e-06	2	1	0.0	128
0.038946	32	42	16	13	40	2.02e-06	0	0	0.0	128
0.039224	34	42	17	11	37	1e-06	2	1	0.0	128
0.039324	36	44	14	12	37	1e-06	2	0	0.0	128
0.039343	33	43	18	11	34	1e-06	2	0	0.0	128
0.039451	37	42	11	8	40	1e-06	2	0	0.0	128
0.039517	30	42	17	5	38	1e-06	0	0	0.0	128
0.039525	33	47	14	12	36	1e-06	2	0	0.0	128
0.039584	16	29	33	42	24	1e-06	2	1	0.0	512
0.039595	35	46	12	10	39	1e-06	0	0	0.0	128
0.039634	12	32	32	44	26	1e-06	2	1	0.0	512
0.039717	32	41	14	14	40	1e-06	0	0	0.0	128
0.039807	15	27	27	43	23	1e-06	2	1	0.0	512
0.039846	12	29	32	47	21	1e-06	2	1	0.0	512
0.039890	31	40	17	9	36	1e-06	0	0	0.0	128
0.039894	21	29	31	49	23	1e-06	2	0	0.0	512
0.039985	17	29	28	46	19	1e-06	0	1	0.0	512
0.040023	17	28	31	46	24	1e-06	0	1	0.0	512
0.040090	34	43	15	9	36	1e-06	0	0	0.0	128
0.040092	13	33	49	46	18	0.007224	0	0	0.057640	256

0.040166	30	44	15	8	37	1e-06	0	0	0.0	128
0.040261	32	42	15	13	37	1e-06	0	0	0.0	128
0.040368	14	34	49	47	19	0.007239	0	1	0.070291	256
0.040557	11	29	31	42	23	1e-06	0	1	0.0	512
0.040583	33	43	13	6	39	1e-06	0	0	0.0	128
0.040691	21	27	31	47	19	1e-06	0	1	0.0	512
0.040737	13	28	30	45	24	1e-06	0	1	0.0	512
0.040795	13	26	26	46	26	1e-06	0	1	0.0	512
0.040918	13	25	24	42	26	1e-06	0	1	0.0	512
0.045045	33	46	15	11	39	0.002275	2	0	0.264817	128
0.047440	11	35	49	46	17	0.007570	0	0	0.165438	256
0.048401	31	43	17	8	39	1e-06	0	0	0.147656	128
0.049623	31	44	12	10	40	2.35e-05	2	0	0.310702	128
0.050411	32	43	12	10	38	0.002341	2	0	0.432325	128
0.050497	35	43	14	9	41	2.98e-05	2	0	0.193868	128
0.063438	9	35	49	45	19	0.008430	0	0	0.318482	256
0.063845	8	35	8	34	25	0.008539	1	1	0.314492	512
0.064028	8 0	37	49	45	14	0.007805	1	0	0.280808	256
0.065928	14	28	26	46	28	1e-06	1	0	0.0	512
0.075260	28	31	16	40	12	0.004324	0	1	0.403866	512
0.079563	30	47	3	35	29	0.006733	1	1	0.256143	128

0.082729	5	38	49	44	15	0.008767	0	0	0.465065	256
0.086085	23	45	36	31	41	0.002386	0	1	0.757406	64
0.092507	7	43	9	31	29	0.008595	1	0	0.619949	512
0.099071	25	16	5	32	2	0.005456	2	0	0.698341	128
0.111001	45	19	6	39	12	0.008571	1	1	0.801601	256
0.115073	50	1	49	31	3	0.008314	2	1	0.442067	64
0.115347	12	38	8	31	29	0.009954	1	0	0.723308	512
0.118661	34	11	10	12	24	0.001924	0	1	0.522646	512
0.121246	17	41	8	2	12	0.000177	1	0	0.435642	640
0.123944	47	49	26	3	26	0.002863	0	1	0.523044	256
0.128675	45	19	6	20	20	0.009632	1	1	0.846676	512
0.142495	16	14	22	12	30	0.002063	0	1	0.874191	1024
0.391099	13	29	37	48	25	1e-06	2	0	0.9	512
0.485474	19	28	25	49	21	1e-06	2	1	0.9	512
0.493399	17	34	35	50	17	0.001101	1	1	0.0	512
0.493399	11	26	27	49	23	0.01	1	1	0.0	512
0.493399	13	35	49	47	14	0.006523	1	1	0.005603	256
0.493399	33	43	18	11	41	0.01	2	1	0.9	128

Table D.2: Sorted data of all evaluations

Bibliography

[WBM12] Nicolas Watrin, Benjamin Blunier, and Abdellatif Miraoui. *Review of adaptive systems for lithium batteries State-of-Charge and State-of-Health estimation*. 2012.

[GM17] Gevorg Gharehpetian and Mohammad Mousavi. *Distributed Generation Systems. Design, Operation and Grid Integration*. Wiley, 2017. Chap. 7.

[1] 1. *State of health*. `https://en.wikipedia.org/wiki/State_of_health`.

[Bac+17] Ines Baccouche et al. *Implementation of an Improved Coulomb-Counting Algorithm Based on a Piecewise SOC-OCV Relationship for SOC Estimation of Li-Ion Battery*. 2017.

[MG] Martin Murnane and Adel Ghazel. *A Closer Look at State of Charge (SOC) and State of Health (SOH) Estimation Techniques for Batteries*. `https://www.analog.com/media/en/technical-documentation/technical-articles/a-closer-look-at-state-of-charge-and-state-health-estimation-techniques.pdf`. Accessed: 2021-08-24.

[SG07] B. Saha and G. Goebel. *Battery Data Set*. NASA Ames Prognostics Data Repository from NASA Ames Research Center, Moffett Field, CA, `https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/`. 2007.

[SS97] J. Sevilla and J. Sola. *Importance of input data normalization for the application of neural networks to complex industrial problems*. 1997.

[BH01] Imad Basheer and M.N. Hajmeer. *Artificial Neural Networks: Fundamentals, Computing, Design, and Application*. 2001.

[Rip96] Brian D. Ripley. *Pattern Recognition and Neural Networks*. 1996.

[Nag+21] Tal Nagourney et al. *The Implications of Post-Fire Physical Features of Cylindrical 18650 Lithium-Ion Battery Cells. Fire Technol 57, 1707–1722 (2021). https://doi.org/10.1007/s10694-020-01077-8*. 2021.

[Gro] René Groiß. *The influence of temperature on the operation of batteries and other electrochemical energy storage systems*.

[HG17] Liang He and Kang G. Shin. *How Long Will My Phone Battery Last*. 2017.

[Zho+21] Wenlu Zhou et al. *Review on the Battery Model and SOC Estimation Method*. 2021.

[XZa19] Bizhong Xia, Jie Zhou, and et al. *A Novel Battery Equalization Method Base on Fuzzy Logic Control Considering Thermal Effect*. 2019.

[Bla+13] Cecilio Blanco et al. *Support Vector Machines Used to Estimate the Battery State of Charge*. 2013.

[Tin+14] T.O. Ting et al. *State-Space Battery Modeling for Smart Battery Management System*. 2014.

[CF10] Mohammad Charkhgard and Mohammad Farrokhi. *State-of-Charge Estimation for Lithium-Ion Batteries Using Neural Networks and EKF*. 2010.

[Mon11] Corey Montella. *The Kalman Filter and Related Algorithms: A Literature Review*. 2011.

[3] 3. *TensorFlow*. `https://en.wikipedia.org/wiki/TensorFlow`.

[4] 4. *TensorFlow API*. `https://www.tensorflow.org/api_docs/`.

[Rei+21] Gonçalo dos Reis et al. *Lithium-ion battery data and where to find it*. 2021.

[5] 5. *Phillip Kollmeyer*. `https://data.mendeley.com/datasets/wykht8y7tg/1`.

[Che+18] Ephrem Chemali et al. *State-of-charge estimation of Li-ion batteries using deep neural networks: A machine learning approach*. 2018.

[Kol+20] Phillip Kollmeyer et al. *LG 18650HG2 Li-ion Battery Data and Example Deep Neural Network xEV SOC Estimator Script*. 2020.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[Dem20] Thomas Demmig. *Praxiseinstieg Machine Learning mit Scikit-Learn, Keras und TensorFlow : Konzepte, Tools und Techniken für intelligente Systeme. Aktuell zu TensorFlow 2*. 2020.

[Ari+16] Vitor Torquato Arioli et al. *A new approach to estimate SoH of lead-acid batteries used in off-grid PV system*. 2016.

[6] 6. *Gradient*. `https://en.wikipedia.org/wiki/Gradient`.

[ASR20] Gabriel C. S. Almeida, A. C. Zambroni de Souza, and Paulo F. Ribeiro. *A Neural Network Application for a Lithium-Ion Battery Pack State-of-Charge Estimator with Enhanced Accuracy*. 2020.

[KB17] Diederik P. Kingma and Jimmy Lei Ba. *ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION*. 2017.

[Nwa+18] Chigozie Enyinna Nwankpa et al. *Activation Functions: Comparison of Trends in Practice and Research for Deep Learning*. 2018.

[CUH16] Djork-Arn´e Clevert, Thomas Unterthiner, and Sepp Hochreiter. *FAST AND ACCURATE DEEP NETWORK LEARNING BY EXPONENTIAL LINEAR UNITS (ELUS)*. 2016.

[Sri+14] Nitish Srivastava et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014.

[7] 7. *Dropout layer*. `https://keras.io/api/layers/regularization_layers/dropout/`.

[8] 8. *Bayesian Optimization Primer*. `https://en.wikipedia.org/wiki/Hyperparameter_optimization`.

[BYC13] J. Bergstra, D. Yamins, and D. D. Cox. *Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures*. 2013.

[MVH21] Lorenzo Maggi, Alvaro Valcarce, and Jakob Hoydis. *Bayesian Optimization for Radio Resource Management: Open Loop Power Control*. 2021.

[Fra18] Peter I. Frazier. *A Tutorial on Bayesian Optimization*. 2018.

[BCF10] Eric Brochu, Vlad M. Cora, and Nando de Freitas. “A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning”. In: *CoRR* abs/1012.2599 (2010). arXiv: `1012.2599`. URL: `http://arxiv.org/abs/1012.2599`.

[SLA12] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. *PRACTICAL BAYESIAN OPTIMIZATION OF MACHINE LEARNING ALGORITHMS*. 2012.

[Sha+16] Bobak Shahriari et al. *Taking the Human Out of the Loop: A Review of Bayesian Optimization*. 2016.

[Gar+14] Jacob R. Gardner et al. *Bayesian Optimization with Inequality Constraints*. 2014.

[UB21] Juan Ungredda and Juergen Branke. *Bayesian Optimisation for Constrained Problems*. 2021.

[9] 9. *Gradient*. `https://nbviewer.org/github/SheffieldML/GPyOpt/blob/devel/manual/GPyOpt_constrained_optimization.ipynb`.